

Inference of Enumerated Types In Java

Scott Arbeitman (Student ID: 181619)

February 17, 2004

Abstract

The release of Java 1.5 adds a popular programming feature – enumerated types. This paper presents an algorithm for inferring enumerated types for Java 1.5 from final static fields in legacy Java code. This process is divided into three phases. First, candidate enumeration constants are selected from the set of static, final fields in the input set of Java classes. Then, typed entities (fields, locals, methods, and formals) in the source code are analyzed to determine if they are suitable candidates for an enumeration type. Finally, Java code is generated with the new enumerated declarations and the new type is inserted into the code, where applicable.

1 Introduction

Java 1.5 introduces many new features to simplify programming and make compile-time checking more robust. Among the new features are: generic types, autoboxing/unboxing of primitive types, `static imports`, enumerated types, and an enhanced `for` loop [5, 4]. Already work has begun inferring these properties from old Java code, and converting older code to new. For example, research is well underway inferring generic types from Java bytecode and outputting new classes that support generic types [6, 2]. Some additions, such as `static import`, are trivial to infer. Others, such as retroactively adding type parameters to generic classes, are more complex.

Enumerated types are a common programming paradigm found in many imperative languages, most notably C++, after which Java is based. Other languages supporting enumerated types are Ada 95 and Pascal. Functional languages, such as Haskell, offer similar functionality vis-à-vis simple parametric polymorphism [8].

Without enumerated types in Java, programmers have adopted a number of methods of emulating them. Many are inconvenient, and most are not type-safe.

One approach is to create a class that represents the enumerated type, while declaring the constructor `private` (figure 1). Each possible value of that type is declared as a `final, static` field within that class. Since the constructor is not visible to any other components, one can be sure there is only one instance of each constant. Any attempt to modify a constant will result in a compile-time error, as desired. This pattern is known as the *typesafe enum pattern*, but it is not widely known or used [3].

Most code (including that in the Java library) uses a sequence of `final, static ints` or `Strings` to denote possible constants (figure 2). Problems that occur with this and other approaches to approximating enumerated types are very similar to those incurred in the absence of generic types [6] and include:

1. **Errors are not Detected During Compilation** Without enumerated types, there are effectively an infinite number of values that any type can range over. These can be primitives and objects. The type-checker does not determine if assigned values or return types are within a certain range of values for a given type. For example, there is nothing preventing a programmer from using the literal 127 in place of one of the color constants in figure 2. Likewise, nothing prevents `BLUE` from being parsed when an integer is required.

```

public class Color {
    private final String name;

    private Color(String name) { this.name = name; }

    public String toString() { return this.name; }

    public static final Color RED = new Color("red");
    public static final Color GREEN = new Color("green");
    public static final Color BLUE = new Color("blue");
}

```

Figure 1: The type-safe enum pattern

```

public class UsesColor {
    public static final int RED = 0;
    public static final int GREEN = 1;
    public static final int BLUE = 2;
    . . .
}

```

Figure 2: The int pattern

2. **Poor Documentation** Pseudo-enum types do not fall under the umbrella of a certain type; therefore there are no formal connections between logical groups of constants (except perhaps their names). Therefore, the programmer’s intention when using these constants may not be apparent. Moreover, constants can be used in mixed contexts, further complicating the code. For instance, one may perform arithmetic on them. Enumerated types explicitly group values under one logical umbrella and those values cannot be used in mixed contexts. Furthermore, the `toString` method for an `enum` type returns a more meaningful `String` than a primitive and some objects. This can make programs easier to test and debug. The default `toString` method returns the name of the constant (e.g. “RED”), which is much easier to understand than “0”.
3. **Performance** Some pseudo-enum approaches (such as having a wrapper class for an enum type, with special `get`, `set`, and `equals` methods that enforce correct usage) do so at a significant cost, both in terms of programmer productivity and execution time. Java enumerated types have performance comparable to integer constants [5].

This paper presents an algorithm for inferring enumerated types from fields in Java based on the more common and least type-safe pattern, such as that in figure 2. While the new `enum` construct is based on the *typesafe enum pattern*, it is more practical to consider the `int` pattern, since it is more common and not type-safe.

The analysis requires as input a set of Java source files. We call these source files “the program”, although there are other dependencies which may constrain the program, such as imported types, which are not included in the analysis. We do not consider any details at the bytecode level; source code refers only to high-level Java code.

Where possible, methods for dealing with external classes and interfaces, such as those prescribed in the libraries, are presented. We consider these files external to the program and we do not require the analysis to alter or observe them in any way.

Enumerated constants are the values that an enumerated type can be assigned. An `enum` type is a standard language type that can only take on values from the set of enumerations. For example, in the enumeration

```

public class ColorUser {
    public static final int RED = 0;
    public static final int GREEN = 1;
    public static final int BLUE = 2;
    public static final int BLACK = -1;
    public static final int WHITE = 1;
    protected int color;
    public ColorUser(int color) { this.color = color; }
    public int getColor() { return this.color; }
    public void setColor(int newColor) { this.color = color; }
    public boolean isRed() { return (color == RED); }
    public boolean isBlue() { return (color == BLUE); }
    public boolean isGreen() { return (color == GREEN); }
    public int newGrey(int black_value, int white_value) {
        return (black_value * BLACK) + (white_value * WHITE);
    }
}

```

Figure 3: A simple example which uses the `int` pattern

```
enum Color {RED, GREEN, BLUE}
```

`Color` is the enumeration type; `RED`, `GREEN`, and `BLUE` are its possible enumerations (enumeration constants). This paper, for brevity, refers to enumeration types as *enum types* while *enum constants* are the possible values of an `enum` type, i.e. those values which it can enumerate over. Enum constants are said to be *related* if they are part of the same enum type. Enum constants are *bound* to their types and cannot occur in other enum types.

A brief overview of the analysis is outlined in the following section. Some key terminology is introduced. The selection criteria for `enum` constants are presented in section 3. The algorithm for deducing the enumerated types are presented in section 4. Section 5 discusses the procedure for outputting new code, including heuristics for naming the new types. In section 6 we explore an alternative approach to outputting code. Finally, section 7 concludes.

A tool was developed to perform the analysis discussed in this paper, albeit with a restricted subset of Java. The subset of Java is based on the JOOS language [1], with additions. Decisions made when designing the tool are mentioned in various parts of the paper, however the tool still remains incomplete.

We shall use a simple running example (figure 3) to illustrate some issues in the paper. When the analysis is complete, we will translate that code into that of figure 4.

2 Overview

Everything that is typed in Java can be given an `enum` type instead of a reference or primitive type, with the exception of things that are thrown (i.e. `Errors` and `Exceptions`). This restriction is simply due to the fact that `enum` types cannot extend other classes, and thus they cannot, in particular, extend `Throwable`. This leaves fields, formals, locals, and methods' return types as candidates for `enum` types. When context is not important, these typed entities are referred to as *enumerables*, since there exists the possibility of having their values enumerated. As the analysis progresses, certain enumerables will be found unsuitable candidate for a type change. When analysis completes, it does not modify the type declaration of these enumerables. We denote enumerables that are not modified as a special type, σ .

When analysis starts, each enumerable is associated with one (nameless) `enum` type, called its *candidate type*. We use the symbol τ to denote the candidate type. τ is fully described by the set of constants it encapsulates or *binds*, and is thus really just a set. We aim to bind constants to

```

public class ColorUser {
    public enum enum_1 { red, green, blue }
    public static final int WHITE = -1;
    public static final int BLACK = 1;
    protected enum1 color;
    public ColorUser(enum_1 color) { this.color = color; }
    public enum_1 getColor() { return this.color; }
    public void setColor(enum_1 newColor) {
        this.color = newColor;
    }
    public boolean isRed() {
        return this.color == enum_1.blue;
    }
    public boolean inGreen() {
        return this.color == enum_1.green;
    }
    public boolean isBlue() {
        return this.color == enum_1.blue;
    }
    public int newGrey(int black_value, int white_value) {
        return (black_value * BLACK) + (white_value * WHITE);
    }
}

```

Figure 4: Modified ColorUser class

each candidate `enum` type within the program, fully describing the new type. A variable's actual declared type, such as `String` or `int` is not important to the analysis.

Each constant in a type τ has τ as its implicit type. This is analogous to the typing of manifest constants. For example, the literal `1` has type `int` and `"Hello"` has type `String`. We want to determine that, for example, `BLUE` has type `Color`.

In summary, we have the following properties:

- Each enumerable ε has an associated candidate `enum` type τ . We denote this relationship as $\varepsilon:\tau$.
- Each type τ binds constants $c_0 \dots c_n$. We use standard set notation in this case: $c_i \in \tau$ since τ is essentially a set of constants. Also, each constant $c_0 \dots c_n$ has a type τ , likewise denoted $c_i:\tau$.
- An enumerable that is considered unsuitable for enumerated and thus does not have its type modified, is denoted as a special type, σ .

When analysis begins, we have

- $\forall \varepsilon \in P \exists \tau : \varepsilon:\tau \wedge \tau = \emptyset$ Each typed entity is enumerable, but only binds the empty set.
- $\forall c \in P \exists \tau : c:\tau \wedge c \in \tau$ Each constant is enumerable, and binds the singleton set, of which it is the only member.

where P is the program being analyzed. Note that σ is not present when the analysis begins; everything is a candidate for an `enum` type.

When analysis concludes on the `ColorUser` class, analysis will determine that, among other things:

- RED: `enum_1` \wedge GREEN: `enum_1` \wedge BLUE: `enum_1`

- `color: enum_1 ∧ getColor: enum_1 ∧ newColor: enum_1`
- `WHITE: σ ∧ BLACK: σ`

We could then output the new class in figure 4. Note that `BLACK` and `WHITE` are not altered, since inferred to be of type σ when arithmetic is performed on them.

3 Collecting Constants

Our initial goal is to determine a set of values that model or approximate the `enum` constants of (possibly several) `enum` types. We call each member of this set a *candidate constant* and each `enum` type will contain values (constants) whose origins are these candidate values. These values are selected from the set of fields in the program, with restrictions. These restrictions are enumerated below.

3.1 Static Properties

Candidate values are chosen from the pool of static fields (sometimes referred to as *class fields* [11]) because they behave predictably when compared using “`==`”. By predictable, we mean two variables return true when compared using “`==`” after both being assigned the same static field. This is opposed to *instance fields* that vary from one instance of an object to the next and, in general, do not have predicable equality comparison. We use the term *reference equality* when describing comparisons of reference types and *object equality* when describing the comparison of classes using the `equals` methods. Comparison of boxed values and enum constants is called *value comparison*. Java uses the same symbol to denote value and reference equality (and inequality) comparison.

Simple equality comparison is a primary benefit of `enum` types. Essentially, equality is “given”; one does not have to implement a unique `equals` method for each new type; it is sufficient (and equivalent) to use “`==`”. This convenience puts a subtle restriction on choosing candidate constants. If constants a and b are equal (`==`), they must be the same variable. Else, they must be rejected as `enum` constants.

I am not happy with the previous paragraph, especially the last two sentences.

In contrast, for a set of static fields in Java, we have reference equality precisely when one *object* references the other, but in the case of primitives, two elements can be equal even if they reference different values (figure 5). Further complicating the issue of equality are `Strings`. When a programmer invokes `string.intern()`, it allows the use of referential equality in place of object equality. This exceptional condition is handled by another consideration altogether, and is discussed in the section 3.2.

We must ensure that each `enum` type does not bind two constants that originate from equal (referential or value) fields. Since we cannot determine which constant “really” belongs to that type, we spoil the type to which both constants are member. We have

$$\forall a, b : a \in \tau \wedge b \in \tau \wedge a == b \longrightarrow \tau \equiv \sigma$$

Although candidate values must be class fields, they need not be explicitly declared `static`. Fields declared within interfaces are implicitly `static` and maintain the above property of predictable equality comparison. Hence, they are suitable candidates for enumeration constants. Each class that implements an interface with fields has access to those fields, but cannot alter their values. This is often used to allow access to constants, without fully qualifying their name (e.g. `Math.PI` vs. `PI`). Java 1.5 solves this problem of added verbosity with `static imports`.

3.2 Finality of Fields

Enum constants are not assignable, they are simply bound to a value at compile-time. We wish to emulate this behavior, but fall a bit short. Java does not offer true constants, only single-assignment fields, the difference being that constants are never `null`. Fields are single-assignable

```

...
public static final int PURPLE = 5;
public static final int VIOLET = 5;
...
public static final State STATE_ONE = new State();
public static final State STATE_TWO = new State();
...
public static final String HI = "Aloha";
public static final String BYE = "Aloha";
...
\\initialize later . . .
public static final Error ERROR_1;
public static final Error ERROR_2;
if (PURPLE == VIOLET) { //this is ALWAYS true
...
if (STATE_ONE == STATE_TWO) { //this is NEVER true
...
if (HI == BYE) { // this is SOMETIMES true
...
if (ERROR_1 == ERROR_2) { // this is SOMETIMES true
...

```

Figure 5: Problems of Reference Equality.

when they are declared `final`, except for fields declared within interfaces which are implicitly `final` (unless they are explicitly declared `static`). Once final fields are bound to a value, they cannot be changed via assignment. However, final reference fields are **not** immutable, since they can still be modified via message passing. We thus insist that fields that have their state changed via message passing be spoiled (exceptions to this rule are discussed in section 4.2).

This handles the `intern` method for `Strings` mentioned above, since it is simply a method call.

There is an alternative approach that was considered [11]. It is static, flow-sensitive analysis that determines which fields, static or virtual, are immutable. This would expand immutability to include not only state-change via assignment, but also detecting changes from all variables reachable from it. Using this analysis would not require a field to be declared `final` since analysis may determine it to be single-assignable anyway. This approach, while interesting, is not adopted because ultimately, programmers use the `int` pattern or use fields from interfaces.

There is always the possibility that a field was never initialized. These fields must be rejected because of the problems of equality comparison when fields are `null`, i.e. if two fields are `null`, they have referential equality.

3.3 Special Consideration of Types

There are cases which meet the aforementioned criteria, but one still may not want to consider them candidate values. Several special cases are discussed below, and generalizations are drawn.

Whether one wishes to include the special cases in analysis does not affect the procedure of this paper. One can simply choose whether to treat these values as `enum` constant candidates or not. It is likely that analysis will exclude these values later.

Arrays

It is possible to have final, static arrays used as an `enum` constant. Common sense may indicate that the programmer did not intend such fields to be used in an enumeration. While including or

```

...
if (Constants.TRUE) { //spoil TRUE
...
while (Constants.USE_DATA && Constants.CONTINUE) //spoil USE_DATA and CONTINUE

```

Figure 6: Boolean variables as constants

excluding them from the analysis is a matter of contention, practically speaking, they do not alter the analysis in any way.

A programmer may also want to treat a single array as an `enum` type, with its entries as its bound constants. This case is not currently handled by the analysis. As is, array access spoils the array as constant and we do not consider converting a single array into an `enum` type, only an `enum` constant.

Throwables

Here again, it is difficult to infer what the programmer had intended by declaring a `Throwable` (or one of its subclasses) as `final` and `static`. It seems logical that the intention was to use this value in a `throw` statement somewhere in the program. `enum` types cannot be subclasses of other types and, in particular, they cannot subclass `Throwable` (see above). Thus candidate `enum` types can never be thrown using the keyword `throw`. Methods for handling a `throw` expression if `Throwables` are considered candidate constants are similar to other operations that cannot occur on `enum` types, such as simple arithmetic. In general, they are spoiled.

The problems of picking candidate `enum` values stem from trying to understand the programmer's intent. Since these special cases will probably not be used in the same context as other constants, analysis will weed them out in later stages.

Booleans

Boolean variables should never be used as `enum` constants the way `int` and `Strings` are (figure 6). If this does occur, we must exclude them as candidate under certain conditions. For example, boolean values are required in contexts where other types are prohibited (e.g. `if` and `while` statements), and one can never substitute an `enum` into these constructs.

3.4 Other Possible Criteria

It is possible to develop more stringent selection criteria for `enum` constant candidates, based on actual implementation of `enum` types in Java 1.5. In particular:

- `enum` types cannot subclass other classes¹, nor can they be extended. Therefore, one can argue that all `enum` constant candidates must extend `Object` directly.
- `enum` types are automatically `Comparable` and `Serializable`. Likewise, we could insist that constants implement these interfaces.
- `enum` constants must not be `Cloneable` since one may not clone an `enum` type. If a constant implements `Cloneable`, it should be spoiled.

These additional criteria are rejected since they effectively exclude primitives and needlessly restrict the pool of constants. Furthermore, while some classes may be `Cloneable`, they may not utilize this functionality. If they do, they are spoiled by the analysis in the next phase.

Finally, there is the consideration of visibility, i.e. is a field `public`, `protected`, or `private`? Because `enum` types can have their own visibility, visibility of constants is not relevant. We simply

¹`enums`, like other reference types, automatically extend `Object`. The actual implementation of `enum` types requires a primordial `java.lang.Enum` class, which is the superclass of all `enum` types. Since this type is not present in earlier Java, there is no reason to consider it.

```

public class Directions {
    public static final int NORTH = 0;
    public static final int SOUTH = 1;
    public static final int EAST = 2;
    public static final int WEST = 3;

    protected static final int RIGHT = 0;
    protected static final int LEFT = 1;
}

```

```

public class Directions {
    public enum enum_1 { NORTH, SOUTH, EAST, WEST }
    protected enum enum_2 { RIGHT, LEFT }
}

```

Figure 7: Mixed visibility of fields

use the same visibility modifier of the constants in that type, with the assumption that all the visibility of each constant is the same (figure 7). If they are not, the type must be spoiled.

4 Binding Constants to Types

At this point, we have a collection of potential constants for `enum` types, and they each belong to a different type τ . We also have for each local, formal, field, and method, an associated candidate `enum` type. In the former case, τ binds a single constant, in the latter, the empty set. In the `ColorUser` class, for each constant, we have:

- RED: $\tau_R \wedge \tau_R = \{\text{RED}\}$
- GREEN: $\tau_G \wedge \tau_G = \{\text{GREEN}\}$

and so on.

Our goal is to merge types τ_i and τ_j when they are used in the same context. We define an operation \sqcup (merge) thus:

$$\tau_i \sqcup \tau_j = \begin{cases} \sigma & \text{if } \tau_i = \sigma \text{ or } \tau_j = \sigma \\ \tau_i \cup \tau_j & \text{otherwise} \end{cases}$$

where \cup is the standard union of sets.

Within the program, we infer type merging in three contexts:

1. Two enumerables, ε_i and ε_j are used in the same context. We merge their types.
2. Two constants, c_i and c_j are used in a way that indicates they are the same type. For this particular case, only equality (inequality) comparison will not spoil the type.
3. One constant and one enumerable are used together. Again, we merge their types.

Because it is not significant which of the above cases occurs, we shall use the symbols α and β to denote an enumerable or a constant being used in the program.

4.1 Notation

We use type inference notation similar to [9, 10]. In this analysis, the type of each variable is given as its declared type; we are not inferring it. What we are inferring is the candidate `enum` type, and its associated constants. The notation used is outlined below.

Methods

We use the notation: $M_{\tau_1, \tau_2, \dots, \tau_n}^\tau$ to denote a method M with return type τ , and formal arguments of type $\tau_1, \tau_2, \dots, \tau_n$. The method has arity n . We denote an invocation of method M with a lowercase m . The type of m is its return type, τ . The type of each formal corresponds to the signature of M , subject to standard Java type rules. Constructor declarations and invocations are analogous to `[void]` methods, and are not handled separately.

Polymorphic Properties

If a method M in class C overrides a method M' from class C' , we write $M \triangleleft M'$. Note that M' can be an abstract method in an abstract class C' . Furthermore, C' can be an interface, which declares a method signature M' . We do not differentiate between abstract methods, interface methods, and concrete methods. We further do not distinguish between `static` methods, and virtual methods.

When instantiating a class C , we denote this as `new C τ_1, \dots, τ_n` or simply C when arguments are not important.

If a class C extends a class C' not included in the analysis, we consider O to be *external* to the analysis. We denote this by $\|C\|$.

Manifest Constants

A manifest constant is a built-in feature of any language. Each manifest constant has a default types, which cannot be a user-defined type. Examples of manifest constants in Java are `1` with type `int`, `1.0` with type `float`, `false` with type `boolean`, and `"Hello World"` with type `String`. We denote a manifest constant as θ .

Inference Rule Notation

Each rule be will of the form:

$$\frac{STATEMENT_1 \quad STATEMENT_2 \quad \dots \quad STATEMENT_m}{CONCLUSION_1 \quad CONCLUSION_2 \quad \dots \quad CONCLUSION_n}$$

Each statement and conclusion may be preceded by L, C, M, V , such as

$$L, C, M, V \vdash E: \tau$$

to indicate that

In class library L , in class C , in current method (constructor) M , with variables V , analysis has determined that E has type τ .

When context is not important, we can simply use $P \vdash S$ to indicate “within the program, S has occurred”.

When single quotes are used, it denotes the operator’s use in Java. For example `'=` denotes assignment in Java while `'+` denotes either `String` concatenation or addition.

4.2 Constraint Rules

We now describe the situations where merging occurs. We accomplish this using a constraint-based inference system [12], based on the constraints below.

$$\text{Assignment} \quad \frac{P \vdash \alpha: \tau_i \quad \beta: \tau_j \quad \alpha' = \beta}{P \hat{\tau} = \tau_i \sqcup \tau_j \quad \alpha: \hat{\tau} \quad \beta: \hat{\tau}}$$

If one item is assigned to the other, they must have the same type.

$$\text{Equality Comparison} \frac{P \vdash \alpha: \tau_i \quad \beta: \tau_j \quad \alpha' ==' \beta}{P \vdash \hat{\tau} = \tau_i \sqcup \tau_j \quad \alpha: \hat{\tau} \quad \beta: \hat{\tau}}$$

When a programmer compares two values for equality, there must exist the possibility that the values can be equal. Thus, the set of constant for each type must be the same.

$$\text{Inequality Comparison} \frac{P \vdash \alpha: \tau_i \quad \beta: \tau_j \quad \alpha' !=' \beta}{P \vdash \hat{\tau} = \tau_i \sqcup \tau_j \quad \alpha: \hat{\tau} \quad \beta: \hat{\tau}}$$

The same is true for inequality, i.e. there must be a possibility that the values are equal, else it would be a tautology.

$$\text{Method Return Types} \frac{L, C, M, V \vdash M: \tau_i \quad \beta: \tau_j \quad \text{return}(\beta)}{P \vdash \hat{\tau} = \tau_i \sqcup \tau_j \quad M: \hat{\tau} \quad \beta: \hat{\tau}}$$

A method's return type is fully specified by the values of the expressions following the **return** keyword. Valid Java ensures a matching of these types.

$$\text{Method Invocations} \frac{P \vdash M_{\alpha_i: \tau_1, \alpha_1: \tau_2, \dots}^{\tau} \quad m_{\beta_1: v_1, \beta_2: v_2, \dots}^v}{P \vdash \hat{\tau}_i = \tau_i \sqcup v_i \quad \alpha_i: \hat{\tau}_i \quad \beta_i: \hat{\tau}_i \quad m: \sigma}$$

Arguments which map to formals indicate that they have the same type, or are assignment compatible. This is a standard Java rule. Therefore, if an argument is an **enum** type, its corresponding formal must be as well. Enumerables that have messages passed to them are spoiled, since, in general, there are no virtual methods associated with each **enum** type. We exempt the **compareTo** method from this rule, since **enum** type are comparable by default and we will sort the constant when outputting each **enum** type. We do not exclude the **toString** method, because we must ensure that the same **String** is returned as in the original program. Similar reasoning applies to all methods that derive from `java.lang.Object`.

$$\text{Method Overriding} \frac{L, C, M, V \vdash M_{\alpha_i: \tau_1, \alpha_1: \tau_2, \dots: \tau_i}^{\tau'} \quad M_{\beta_i: v_i, \beta_j: v_j, \dots}^{v'} \quad M \triangleleft M'}{P \vdash \hat{\tau}_i = \tau_i \sqcup v_i \quad \alpha_i: \hat{\tau}_i \quad \beta_i: \hat{\tau}_i} \\ P \vdash \phi = \tau' \sqcup v' \quad M: \phi \quad M': phi$$

We must preserve the polymorphic properties of functions, so that when a method is invoked, Java can resolve which method is actually required, from the class hierarchy. Changing the method signatures disrupts this property. Furthermore, Java requires overridden methods with matching formals to have the same return type, so the return types must be the same.

The following rules apply to spoilage:

Manifest Constants $P \vdash \theta: \sigma$

$$\text{External Classes} \frac{L, C, M, V \vdash C \text{ "extends" } ||O|| \quad M_{\alpha_1: \tau_1, \alpha_2: \tau_2, \dots}}{P \vdash M: \sigma \quad \alpha_i: \sigma}$$

$$\text{External Interfaces} \frac{L, C, M, V \vdash C \text{ "implements" } ||O|| \quad M_{\alpha_1: \tau_1, \alpha_2: \tau_2, \dots}}{P \vdash M: \sigma \quad \alpha_i: \sigma}$$

Class Instance Creation $P \vdash \text{"new"} \quad C: \sigma$

$$\text{Exception Handling} \frac{L, C, V, M \vdash \text{"throw"}(X)}{L, C, V, M \vdash X: \sigma}$$

Arithmetic $\frac{P \vdash \alpha \oplus \beta}{P \vdash \alpha: \sigma \quad \beta: \sigma}$ where \oplus are all the Java primitive operations, such as '+', '-', '%', '!', etc.

$$\text{Arrays, Array access, Array assignment} \frac{P \vdash \alpha \text{ "[" } \beta \text{ "]" } = \gamma}{P \vdash \alpha: \sigma \quad \beta: \sigma \quad \gamma: \sigma}$$

$$\text{Casting} \frac{P \vdash \text{"(" } C \text{ ")"} \alpha}{P \vdash \alpha: \sigma}$$

Static methods $\frac{P \vdash \|C\|.m(f_1, f_2, \dots)}{P \vdash f_i: \sigma \quad m: \sigma}$

Static fields $P \vdash \|C\|.a: \sigma$

One additional rule is needed, to specify that a constant must belong to only one `enum` type:

$$\forall a \in P : a \in \tau_i \wedge a \in \tau_j \longrightarrow \tau_i \equiv \tau_j$$

This is a fundamental axiom of an `enum` type.

5 Outputting New Code

We have for each enumerable ε an associated `enum` type. We now wish to output new code, with the original type declaration replaced with an `enum` type. We also wish to insert a declaration for each new `enum` class as an inner class where its constants were defined. Before doing this, though, we must weed out some candidate types.

5.1 Weeding

Some types may not have been spoiled, but are still not candidate for `enum` types. These are:

Empty types The program does not use this variable in any context. It was most likely intended for use by classes external to the analysis.

One-member types These types are not enumerable since they only bind one value. They are therefore simply constants, and we leave them as such.

Mixed-visibility constants These constants cannot be logical groupings, since their visibility is mixed. Modifying the visibility (e.g. making every constant `public`) may have unintended consequences.

Mixed-origin constants These are constants which belong to the same type, but originate from different classes, interfaces, or a combination thereof. Since there is not baseline logical grouping, we do not know to which class an inner `enum` declaration belongs.

Constants that have value or referential equality As discussed in section 3.1, we cannot have two constant within an `enum` that are equal. We must therefore assume this is not a legitimate type.

5.2 Naming New Types

Here, we offer a heuristic for naming the new `enum` types. In many cases, logical grouping of constants will have the same prefix or suffix, usually separated from the rest of the identifier by an underscore. In these cases, we can use this prefix or suffix as the name of the `enum` type. We then drop that prefix or suffix from the candidate constants, and use those names as the `enum` constants (figure 8).

Because each constant has its own name-space within the program, there is no chance of ambiguity when accessing the constant name.

If there is no common suffix or prefix, we must arbitrarily choose a type name such as `Enum_1`.

We wish the outputted types to be in line with Java coding standards. However, there is no unanimity on how to code `enum` types. In C++, the constants are kept as lower case, but it has been proposed [5] to keep the constants of all upper-case. This is the style used in this paper, and the approach taken by the tool.

<pre> public class JOptionPane extends JComponent { ... public static int YES_OPTION = 0; public static int NO_OPTION = 1; public static int CANCEL_OPTION = 2; public static int OK_OPTION = 3; public static int CLOSED_OPTION = 4; ... </pre>	<pre> public class JOptionPane extends JComponent { ... public enum Option { YES, NO, CANCEL, OK, CLOSED } ... </pre>
--	---

Figure 8: Naming heuristic for enum types and their constants (from `javax.swing.JOptionPane`)

<pre> public class Constants { public static final int NORTH = 0; public static final int SOUTH = 1; public static final int EAST = 2; public static final int WEST = 3; } </pre>	<pre> public enum Constants { NORTH, SOUTH, EAST, WEST } </pre>
---	---

Figure 9: Transformation of a class that contains only constants

5.3 Generating Code

Generating code is a two-step process. First, `enum` type declarations are inserted into classes or interfaces in lieu of the subsequent field declarations. Second, for each enumerable which has not been spoiled, its type declaration is replaced with its `enum` type.

There is no need to worry about importing additional classes, as all new `enum` types are within the scope of a previously imported class or each reference to that class is fully qualified. If a class consists only of constants, we replace that class with an `enum` (figure 9).

Enum constants are `Comparable` by default, and this comparison is wholly determined by the ordering of constants. Therefore, constants must be sorted and output in ascending order. If the reference type was not comparable, there is no logical way to group the constants. A good heuristic is to output them in the order they were declared. We consider default comparison a limitation of the `enum` pattern, since not all constants can be logically ordered. Primitives should be ordered using the '`<=`' operator.

6 Uniqueness of the Enum Pattern

The Java `enum` pattern is more powerful than other `enum` types, such as that in C++. In particular, each `enum` type can contain arbitrary data, including fields and methods. Consequently, they can implement arbitrary interfaces.

This presents an alternative method for inferring enumerated types, and for retaining spoiled types. In fact, there is never a need to spoil anything in the analysis, and everything can become an `enum` type. This is because we can coerce any type to its value using the `enum` pattern, while taking advantage of its extra features. Figure 10 shows an alternative approach to inferring `enum` types from the `ColorUser` class in figure 3. Whenever an `enum` is spoiled, we can simply treat it as an `enum` type, but coerce it to its original value.

In general this approach is not helpful, because it does not offer any of the benefits of a real `enum` type; it just makes code more verbose. However, in the case of spoilage via an enumerable or constant being used as an argument to an external method, it may be preferable to coerce that variable to its value. This would presumably allow more `enum` types to be inferred, especially when library code is not offered as input in the analysis.

```

public class ColorUser {
    public enum enum_1 { RED, GREEN, BLUE }
    public enum enum_2 {
        WHITE(-1), BLACK(1);
        public enum_2(int i) { this.i = i; }
        private int i;
        public int value() { return this.i; }
    }
    protected enum1 color;
    public ColorUser(enum_1 color) { this.color = color; }
    public enum_1 getColor() { return this.color; }
    public void setColor(enum_1 newColor) {
        this.color = newColor;
    }
    public boolean isRed() {
        return this.color == enum_1.blue;
    }
    public boolean inGreen() {
        return this.color == enum_1.green;
    }
    public boolean isBlue() {
        return this.color == enum_1.blue;
    }
    public enum_2 newGrey(int black_value, int white_value) {
        return (black_value * BLACK.value()) + (white_value * WHITE.value());
    }
}

```

Figure 10: Modified ColorUser class using value coercion

<pre> public static final int ONE = 1; public static final int TWO = 2; public static final int THREE = 3; ... for (int i = ONE; i <= THREE; i++) { ... </pre>	<pre> enum Number { ONE, TWO, THREE } ... for (Number n : Number.VALUES) { ... </pre>
---	---

Figure 11: Conversion of for loop to support enum types

6.1 Special Methods

We currently spoil enumerables that access methods external to the program. Alternatively, we can choose to consider certain methods internal, while admitting it might affect the behavior of the program. First, the `compareTo` method could be included since each `enum` type is comparable, by default. If the constants are sorted, the behavior of this method should remain the same.

Second, when coercing an enumerable to a `String`, directly via the `toString` method, or indirectly via the `+` operator, we can choose not to spoil the type. We could further extend this to arguments passed to `System.out.println()`, since they are just coerced to a string as well.

7 Conclusion

This paper presented an algorithm for inferring enumerated types in Java based on the new `enum` construct in Java 1.5. With the retrofitted Java classes, the programmer can better understand the logical grouping of constants, and can more safely reuse older components. Method signatures become more meaningful when return and formal types are replaced by `enum` types.

Programs that implemented constants using resource intensive types, such as `Strings`, should notice an improvement in performance. Furthermore, one can be sure of the validity of each `enum` type's `equals` method since it was done by the compiler. Serialization is also taken of care by the compiler, not the programmer, potentially preventing future troubles.

7.1 Future Work and Limitations

Java 1.5 supports an enhanced `for` loop which can iterate over all elements in a grouping (array members, `Collections`, and `enum` constants). The current analysis will spoil an `enum` type when a programmer attempts to iterate over its constants. It would be helpful if future work could infer that a standard `for` loop was in fact iterating over the entire `enum` type (figure 11).

The current `int` pattern does not allow insertion into collections, since they are not reference types. However, Java 1.5 supports `enum` types as elements in a `Collection`. This offers the programmer additional flexibility, and a better way to group and organize data.

Because `enum` types can be added to collections and used as generic parameters, this research combined with that in [6] could yield generic types that are parameterized by `enum` types. This would produce code that had all the benefits of both generic types and `enum` types. Furthermore, Java 1.5 will offer special classes to handle collections of only enumerated types, which will offer better performance than standard collections. Changing from older classes to special optimized `enum` structures could speed program performance.

This analysis does not, in general, generate correct `enum` types from the *typesafe enum pattern*. Future work should attempt to infer a complete `enum` type, including methods and fields, from that pattern.

Additionally, we considered Java at the source code level. There is no reason to exclude `.class` files from analysis. Deriving properties from bytecode is a common research area [7]. This would further extend the amount of legacy code that could be retro-fitted with `enum` types.

Finally, while the analysis is guaranteed to be type correct within the set of files inputted, one can ensure that all dependencies are accounted for. In particular, objects which have been

serialized and written to a file may not be able to be reinstantiated, since their representation has changed. We offer no method of retrofitting serialized objects in this analysis.

References

- [1] Developing a joos compiler using a sablecc framework. Honor's project, McGill University, 2000.
- [2] Stoler Alan, Cartwright. Efficient implementation of run-time generic types for java. 2002.
- [3] Joshua Bloch. Shift into java. Internet. <http://developer.java.sun.com/developer/Books/shiftintojava/page1.htm> Accessed on 28 November, 2003.
- [4] Gilad Bracha. Adding generic types to java programming language. Jsr014, JavaSoft, Sun Microsystems, 2001.
- [5] Gilad Bracha and Joshua Bloch. Extending the java programming language with enumerations, autoboxing, enhanced for loops, and static import. Jsr 201, JavaSoft, Sun Microsystems, 2002.
- [6] Alan Donovan and Michael D. Ernst. Inference of generic types in java. Technical Report TR-899, Massachussets Institute of Technology, 2003.
- [7] Etienne Gagnon and Laurie Hendren. Intra-procedural inference of static types for java bytecode. Technical Report 5, McGill University, October 1998.
- [8] Cordelia Hall, Kevin Hammond, Simon L. Peyton Jones, and Phillip L. Wadler. Type classes in haskell. In *ACM Transactions on Programming Languages and Systems*, volume 18, pages 109–138, 1996.
- [9] Atsushi Ohori. A simple semantics for ml polymorphism. In *Proceedings of the fourth international conference on Functional programming languages and computer architecture*, pages 281–291, 1990.
- [10] Benjamin C. Pierce and David N. Turner. Local type inference. In *ACM Transactions on Programming Languages and Systems*, volume 22, pages 1–44, 2000.
- [11] Sara Porat, Marina Biberstein, Larry Koved, and Bilha Mendelson. Automatic detection of immutable fields in java. Centre for Advanced Studies on Collaborative research. IBM Centre for Advanced Studies, IBM Press, 2000.
- [12] Tiejun Wang and Scott F. Smith. Precise constraint based type inference in java. Lectures Notes in Computer Science.