Implementing type erasure based on Featherweight Java

Piotr Krzemiński

Abstract

Featherweight Java is a minimal core calculus describing the Java language type system. It provides two calculi named *FJ* – for plain classes with fields and methods, and *FGJ* – extending the first one with generic types. Type erasure is expressed as a translation from *FGJ* to *FJ* that preserves appropriate properties around type-checking and evaluation semantics. In this article we review implementation[1] of these calculi realized in *Scala* and we provide comprehensive examples demonstrating type erasure in action.

1 Introduction

Certain class-based programming languages provide a concept of generic types. It enables parameterizing classes with type parameters. Such a feature makes it possible to write polymorphic code that can work with arbitrary actual type arguments. One of the key applications of generic types can be found in standard libraries, e.g., collections and algorithms working with them. For instance, a sorting algorithm can be implemented as a function taking collection of elements and returning sorted collection, with no particular concern about type of elements inside the collection, as soon as it knows how to compare them.

There are several possible implementations of generics, including:

- **type passing** – it preserves information about type parameters at runtime, which allows to distinguish for example **List<Integer>** from **List<String>**; this implementation is chosen in *.NET* languages like *C#*
- **type instantiating** – for every instantiation of parameterized class with actual type arguments, a separate class is generated that maintains no information about generic types – for example **List$Integer** and **List$String**; we can still distinguish between them, but no type parameter information is present at runtime. This implementation is present in *C++* templates
- **type erasure** – it eliminates information about type parameters at compilation time, replacing them with their so-called *type bounds*; at runtime we have only single **List<Object>** class which can hold any elements; we cannot distinguish lists of integers from lists of strings in this implementation. Type erasure is used by the *Java* language.

In this article we will review implementation of two small programming languages that imitate subsets of Java, being syntactically compatible with the full language, defined in [1]. These two languages are:

- **FJ** – minimal subset of Java with classes, fields and methods only
- **FGJ** – the language extended with type-parameterized classes and methods.

We will define syntax, look at the examples and express type erasure as translation from *FGJ* to *FJ* that preserves some important properties about types and behaviour. We will not explain all the details of provided erasure implementation, but instead will look at the example programs and their erased version to see type erasure in action. Reading the *Featherweight Java* paper is not absolutely required, although highly recommended for readers willing to deeper understand erasure rules, where they are clearly defined and comprehensively explained.

The implementation is written in Scala 2.11. **Java JDK**[2] and **SBT** [3] are required to be installed in order to run the examples.

[1] The implementation is available at `https://github.com/krzemin/type_erasure_featherweight_java`
[2] `http://www.oracle.com/technetwork/java/javase/downloads/index.html`
[3] `http://www.scala-sbt.org`

2 Featherweight Java

We start from introduction of the *Featherweight Java* calculus at the abstract level and see how to encode simple programs in *FJ* and *FGJ*.

2.1 Idea

Looking for a tool to precise describing the Java type system, we need to focus on modelling only those parts of the language which are important from the type system perspective while ommiting those which are not. Trying to model full Java in a formal way would result in an enormous calculus being hard to grasp. Therefore *Featherweight Java* favours compactness over completeness, providing only few combinators, while still being a legal subset of Java, only little larger than the original λ-calculus.

Compactness of the *FJ* is achieved by reducing the language heavily. It comes down to:

- no concurrency primitives like `synchronized` keyword
- no reflection
- no interfaces
- no method overloading
- no inner classes
- no static members
- no member access control – all methods and fields are public
- no primitive types
- no null pointers
- no assignments/setters

Instead, we focus only on minimal language subset, including:

- mutually recursive class definitions
- object creation
- field access
- method invocation, overriding and recursion through `this`
- subtyping
- casting

2.2 Syntax

Let's start with a simple example – an immutable `Pair` class definition.

```
class A extends Object {
  A() { super(); }
}
class B extends Object {
  B() { super(); }
}
class Pair extends Object {
  Object fst;
  Object snd;
  Pair(Object fst, Object snd) {
    super(); this.fst = fst; this.snd = snd;
  }
  Pair setfst(Object newfst) {
    return new Pair(newfst, this.snd);
  }
}
```

FJ is a class-based language where we can define classes like in Java, but satisfying some constraints:

- we always write the super class name, even if it's trivial (`Object`)
- we always write the receiver of field or method, even if it's trivial (`this`)
- `this` is simply a variable rather than a keyword, unlike in full Java
- we always write constructor which initializes all fields defined in that class and call `super` which refers to the super class constructor, which initializes its fields, etc...
- constructors are the only place where `super` or `=` appears

2.2.1 Expressions

In FJ we have 5 types of expressions, which can appear in methods body:

- **variable access** – `newfst` or reference to `this`
- **object construction** – `new A()`, `new B()` or `new Pair(newfst, this.snd)`
- **field access** – in `this.snd` expression a field named `snd` is accessed on the object reffered by a variable `this`
- **method invocation** – `e3.setfst(e4)` is an example of invocation of method `setfst` on object `e3` with single argument `e4`
- **casts** – `(A)(new Pair(new A(), new B()).fst)` is an example of type cast used to recover type information about `fst` field

2.2.2 Programs

In *FJ*, programs consist of a class table and an expression to be evaluated, which corresponds to static `main` method in executable Java classes. We intuitively expect that an expression

```
new Pair(new A(), new B()).setfst(new B())
```

...will eventually evaluate to

```
new Pair(new B(), new B())
```

2.3 Extending with generic types

Let's extend the *FJ* calculus with generic types. We parameterize class `Pair` with two type parameters `X` and `Y`. Using them, we encode types of fields `fst` and `snd`, accordingly.

```
class Pair<X extends Object, Y extends Object> extends Object {
  X fst;
  Y snd;
  Pair(X fst, Y snd) {
    super(); this.fst = fst; this.snd = snd;
  }
  <Z extends Object> Pair<Z, Y> setfst(Z newfst) {
    return new Pair<Z, Y>(newfst, this.snd);
  }
}
```

More generally, the syntax is extended with:

- type parameters lists for classes and methods – in the example above `X` and `Y` are type parameters for class `Pair`, while `Z` is a type parameter of method `setfst`
- every type parameter has to be bounded by some actual class type, possibly parameterized with type variables, e.g., `X extends C<X>`
- in contrast to Java we always write the bound even if it is `Object`
- object construction and method invocation both take type arguments list like `new Pair<Z, Y>(...)` or `.setfst<B>(...)`, but empty parameter lists (`<>`) can be omitted

Our refined example program looks as follows.

```
new Pair<A,B>(new A(), new B()).setfst<B>(new B())
```

And it evaluates to expression

```
new Pair<B,B>(new B(), new B())
```

2.4 Type erasure as translation from FGJ to FJ

We can express type erasure as compilation from *FGJ* syntax to *FJ* by replacing all type variables with their bounds and inserting some number of casts, when needed to smartly recover type information from the original *FGJ* code. The example class `Pair<X, Y>` after erasure looks exactly like the previous `Pair` class without generic types. Similarly, the following expression:

```
new Pair<A,B>(new A(), new B()).snd
```

erases to:

```
(B)new Pair(new A(), new B()).snd
```

Notice that the cast to `B` was inserted to restore type of `snd` field which is annotated with type `Object` in the erased `Pair`.

3 Implementation review

Having gradual introduction to *FJ* and *FGJ* behind, let's get sight of the Scala implementation of these small languages.

3.1 FJ module

FJ-related code is contained in `src/main/scala/fj` directory. There are syntax for *FJ* programs defined in `AST.scala`, type checker in `Types.scala` and evaluator in `Eval.scala`.

3.1.1 Syntax

Classes, fields and methods are represented by following set of case classes.

```
type VarName = String
type TypeName = String

case class Class(name: TypeName,
                 baseClass: TypeName,
                 fields: List[Field],
                 methods: List[Method])

case class Field(name: VarName,
                 fieldType: TypeName)

case class Method(name: VarName,
                  resultType: TypeName,
                  args: List[Argument],
                  body: Expr)

case class Argument(name: VarName,
                    argType: TypeName)
```

We represent *FJ* classes using 4 nested data structures which hold all necessary information about base classe, fields and methods. There are type aliases defined for type and variable names, internally represented as strings. Similarly, we encode expressions using `Expr` trait and following case classes extending it:

```scala
trait Expr

case class Var(name: VarName) extends Expr

case class FieldAccess(expr: Expr,
                       fieldName: VarName) extends Expr

case class Invoke(expr: Expr,
                  methodName: VarName,
                  args: List[Expr]) extends Expr

case class New(className: TypeName,
               args: List[Expr]) extends Expr

case class Cast(className: TypeName,
                expr: Expr) extends Expr
```

In actual implementation all those classes have overriden method `toString` which prettifies syntax of our programs when printing to the console.

3.1.2 Example encoded using Scala syntax

Let's review how we can encode our first example with `Pair` class implementation.

```scala
val A = Class("A", "Object")
val B = Class("B", "Object")
val Pair = Class(
  name = "Pair",
  baseClassName = "Object",
  fields = List(
    Field("fst", "Object"),
    Field("snd", "Object")
  ),
  methods = List(
    Method(
      name = "setfst",
      resultType = "Pair",
      args = List(Argument("newfst", "Object")),
      body = New("Pair", List(
        Var("newfst"), FieldAccess(Var("this"), "snd")
      ))
    )
  )
)
```

It's just straightforward rewriting our `Pair` class with two fields and one method. We represent class tables and programs as follows.

```
type ClassTable = Map[TypeName, Class]
case class Program(classTable: ClassTable, main: Expr)

val classTable: ClassTable = buildClassTable(List(A, B, Pair))
val main: Expr = Invoke(
  New("Pair", List(New("A"), New("B"))),
  "setfst",
  List(New("B"))
)
val program = Program(classTable, main)
```

We have helper function `buildClassTable` which takes list of classes and returns class table built out of them. `Program` is just, following definition, paired class table with main expression to be evaluated.

3.1.3 Type checker

There are type-checking rules provided in *FJ* paper, which are implemented in `fj.Types`.

Subtyping in *FJ* is reflexive and transitive closure of inheritance relation between classes. It can be decided only by looking at class table. Implementation of subtyping is given as recursive function at `fj.Types.isSubtype`.

Main type-checking function is `fj.Types.exprType` which find concrete type of expression in given typing context Γ or indicates that expression is incorrectly-typed. Context Γ contains information about actual types of available variables and is represented as `Map[VarName, SimpleType]`. There is also auxilliary function `fj.Types.progType` which type-checks a whole program, ensuring that all classes, fields, methods are well-typed according to the typing rules, and returns type of main expression.

3.1.4 CBV Evaluator

In [1] there were reduction rules given for expressions in the form of so-called *operational semantics*, which doesn't precise the order of evaluation. Trying to implement expression evaluator, some evaluation strategy have to be chosen. This implementation is realized with *call by value* semantics, which corresponds to that from full Java, where method's arguments are evaluated from left to right.

From the *FJ* calculus point of view when some reduction error occurs (like trying to create object of unknown class or trying to invoke non-existing method), such configuration is called *stuck* and the evaluation cannot be continued. In this implementation we don't bother too much about error handling in the interpreter. When some error configuration is detected, we simply throw `RuntimeException` with appropriate error message, forgetting about result we computed so far.

The evaluator is rather simple adaptation of reduction rules to *call by value* strategy. It can be found at `fj.Eval.evalExpr` for evaluation expressions in given context (i.e. class table) and auxilliary function `fj.Eval.evalProg` which takes program, builds class table and evaluates its main expression.

3.1.5 Running examples

```
println(program.main) // prints: new Pair(new A(), new B()).setfst(new B())
val programT = programType(program) // Some(Pair) - type of main expression
val result = evalProg(program) // New("Pair", List(New("B"), New("B")))
println(result) // prints: new Pair(new B(), new B())
```

Similar example can be run from console by typing:

```
sbt "runMain fj.examples.Pairs"
```

Let's encode some more interesing program in our language. In *FJ* we don't have primitive types, especially numbers. But there is a way to encode natural numbers using just classes and objects, similarly to *Church numerals* in the λ-calculus, but instead of folding functions, we will fold objects of class `Succ` n times over the instance of class `Zero` to represent number n.

```
class Nat extends Object {
  Nat() { super(); }
  Nat succ() { return new Succ(this); }
  Nat plus(Nat n) { return n; }
}

class Zero extends Nat {
  Zero() { super(); }
}

class Succ extends Nat {
  Nat prev;
  Succ(Nat prev) { super(); this.prev = prev; }
  Nat plus(Nat n) { return this.prev.plus(n.succ()); }
}
```

We represent 0 as `new Zero()`, 1 as `new Succ(new Zero())`, 2 as `new Succ(new Succ(new Zero()))`, and so on.

Addition is implemented as recursive function with base case at 0 (indeed, $0 + n = n$). Recursive step is in the class `Succ` – it transforms general addition $m + n$ into $(m - 1) + (n + 1)$ until base case for $m = 0$ is reached. Method `succ` is implemented in class `Nat` as wrapping the actual number `this` into object of `Succ` class.

There is one subtelty connected with implementation of method `plus`. Base case of recursion has to be implemented in class `Nat` to satisfy type-checking of `Succ` class. In full Java we would probably defined this method as abstract in `Nat` class and provide two actual implementations in `Zero` and `Succ`. But in *FJ* we don't have abstract methods and without method `plus` declared in `Nat`, type checking of recursive invocation `this.prev.plus(...)` in `Succ` class would fail.

You can find this example encoded at `fj.examples.Numbers` and run it by typing:

```
sbt "runMain fj.examples.Numbers"
```

3.2 FGJ module

FGJ-related code is contained in `src/main/scala/fgj` directory. There are syntax for *FGJ* programs defined in `AST.scala` and type checker in `Types.scala`.

3.2.1 Types

As types are now part of our classes, methods and expressions AST, let's review them first.

```
type TypeVarName = String

trait Type

case class TypeVar(name: TypeVarName) extends Type

case class ClassType(className: TypeName,
                     argTypes: List[Type]) extends Type
```

We have new type alias `TypeVarName` for type variables (again, internally just strings). We have 2 form of types now: *type variables* and *class types* parameterized by some number of types (which again can be type variables or class types).

3.2.2 Classes

```
case class BoundedParam(typeVar: TypeVar,
                        boundClass: ClassType)

case class Class(name: TypeName,
                 typeParams: List[BoundedParam],
                 baseClass: ClassType,
                 fields: List[Field],
                 methods: List[Method])

case class Field(name: VarName, fieldType: Type)

case class Method(name: VarName,
                  typeParams: List[BoundedParam],
                  resultType: Type,
                  args: List[Argument],
                  body: Expr)

case class Argument(name: VarName, argType: Type)
```

`BoundedParam` corresponds to single `Z extends Object` from our example. It is definition of type variable, bounded by some class type. Notice that we can write recursive type expression in bounds (like `X extends C<X>`) thanks to that definition of class types, which are parameterized with arbitrary types.

Classes are parameterized by a list of *bounded parameters*. Notice change in `baseClass` signature which now is not only name reference, but is class type which can be parameterized with type variables, like in example below.

```
class List<X extends Object> extends Collection<X> { ... }
```

Methods also can be parameterized with type variables. We can use them to encode method's return type and argument types.

`ClassTable` and `Program` definitions are straightforwardly adjusted to use refined types.

3.2.3 Expressions

AST for expressions is mostly unchanged. As before, we have 5 forms of them.

```
trait Expr

case class Var(name: VarName) extends Expr

case class FieldAccess(expr: Expr,
                       fieldName: VarName) extends Expr

case class Invoke(expr: Expr,
                  methodName: VarName,
                  typeArgs: List[Type],
                  args: List[Expr]) extends Expr

case class New(classType: ClassType,
               args: List[Expr]) extends Expr

case class Cast(classType: ClassType,
                expr: Expr) extends Expr
```

The only difference beside type adjustments is in `Invoke` expression which now takes also list of type parameters to be instantiated.

3.2.4 Type checker

In *FGJ* typechecking rules are bit more complicated. First of all, subtyping is not relation between class names any more, but is generalized for all type forms, including type variables. Therefore we differentiate two separate relations:

- **subclassing** – it corresponds to *FJ*'s subtyping, can be decided only using class table
- **subtyping** – generalized relation between all types, can be decided using additional environment Δ which maps type variables to their bounds, where bounds are just class types with actual type arguments given.

Covariant method overriding Unlike to *FJ*, where we allowed method overriding only with corresponding (i.e. identical) signatures, in *FGJ* covariant method overriding on the method's result type is allowed. Result type of a method may be a subtype of the result type of the corresponding method in the superclass, although the bounds of type variables and the argument types must be identical (modulo renaming of type variables).

Function for typing expressions is located at `fgj.Types.exprType`, now it takes expression, class table and two contexts Γ and Δ. Again, we have auxilliary `fgj.Types.programType` which checks also well-typedness of classes and methods.

3.2.5 Excercise: type-passing evaluator

We will not provide *call by value* evaluator for type-passing semantics, leaving it as an excercise for the reader to take *FJ* evaluator code and adjust it to support evaluating programs in syntax with generic types.

Hint As well as we maintain environment for variables and its values, you may need to maintain additional environment mapping type variables to actual class types.

3.3 Type erasure

Erasure-related code is contained in `src/main/scala/erasure` directory.

3.3.1 Overview

The general idea of type erasure is to translate *FGJ* programs into *FJ* ones. To perform that task, we have to define erasure for all parts of our programs. Wanting to adopt erasure rules from [1], several functions are defined, to translate:

- FGJ types to FJ types – `erasure.Erasure.eraseType`
- FGJ expressions to FJ expression – `erasure.Erasure.eraseExpr`
- FGJ classes to FJ classes – `erasure.Erasure.eraseClass`
- and finally we have auxilliary function which merge results and translate a whole FGJ program to FJ program – `erasure.Erasure.eraseProgram`

3.3.2 Examples

Instead of exploring implementation details, let's catch some more interesting *FGJ* programs and their erased versions to see the rules in action.

Example 1 natural numbers revisited
This is extended version of natural numbers implementation in *FGJ*.

```
class Summable<X extends Object> extends Object {
  X plus(X other) { return other; }
}

class Nat extends Summable<Nat> {
  Nat() { super(); }
  Succ succ() { return new Succ(this); }
}

class Zero extends Nat {
  Zero() { super(); }
}

class Succ extends Nat {
  Nat prev;
  Succ(Nat prev) { super(); this.prev = prev; }
  Nat plus(Nat n) {
    return this.prev.plus(n.succ());
  }
}
```

We introduced class `Summable<X>` which have one method `plus`. In Java we would probably make this class an interface, but in *FGJ* we don't have interfaces, so we have to provide default implementation returning some value of type `X`. Fortunately we have parameter of type `X`, so we use it as a return value. It turns out that it is still valid implementation of `plus` for class `Zero`, so we don't have to re-implement it there. We made our `Nat` class a subclass of `Summable<Nat>`. For class `Succ` implementation of `plus` is the same as before. Spot another slight difference in return type of `succ` method in class `Nat` – now it is declared to be `Succ`; we will need that to demonstrate erasure of covariant method overriding in result type in one of the following examples.

Let's use function `erasure.Erasure.eraseClass` to generate erasure for these classes.

```
class Summable extends Object {
  Summable() { super(); }
  Object plus(Object other_) { return other_; }
}

class Nat extends Summable {
  Nat() { super(); }
  Succ succ() { return new Succ(this); }
}

class Zero extends Nat {
  Zero() { super(); }
}

class Succ extends Nat {
  Nat prev;
  Succ(Nat prev) { super(); this.prev = prev; }
  Object plus(Object n_) {
    return (Nat)(this.prev.plus((Nat)(n_).succ()));
  }
}
```

What did the erasure change here?

- in class `Summable` type parameters list was removed and all type variables were replaced with `Object` – which was declared bound for `X` variable (see `X extends Object` in original class)
- class `Nat` now extends our erased `Summable` class
- according to `plus` method signature change in `Summable`, signature of `plus` in `Succ` class was adjusted to be identical (modulo argument names); to recover information about types, two casts to `Nat` were inserted: first over the access to `n_` variable, second over the invocation of method `plus` which happened to return natural number in generic version.

Now, let's construct simple expression using these classes. This will correspond to arithmetic operation $2 + 1$.

```
new Succ(new Succ(new Zero())).plus(new Succ(new Zero()))
```

After erasure it looks almost the same.

```
(Nat)(new Succ(new Succ(new Zero())).plus(new Succ(new Zero())))
```

Now, our `plus` method returns an `Object`, but type erasure was smart enough to insert upcast around invocation of this method, to recover correct type from original program.

Example 2 summable lists

Let's review another example - lists which can contain some summable elements and are able to compute total sum of all their elements.

```
class List<X extends Summable<X>> extends Object {
  List() { super(); }
  X sum(X zero) { return zero; }
}

class Nil<X extends Summable<X>> extends List<X> {
  Nil() { super(); }
}

class Cons<X extends Summable<X>> extends List<X> {
  X head;
  List<X> tail;
  Cons(X head, List<X> tail) {
    super(); this.head = head; this.tail = tail;
  }
  X sum(X zero) {
    return this.tail.sum(zero).plus(this.head);
  }
}
```

We have base `List<X>` class and its two subclasses:

- `Nil` – corresponding to empty list
- `Cons` – list constructor which holds single element `head` of type `X` and rest of list – `tail` of type `List<X>`

For example list `[1, 0]` can be encoded as following expression:

```
new Cons<Nat>(new Succ(new Zero()), new Cons<Nat>(new Zero(), new Nil<Nat>()))
```

Method `sum` takes parameter `zero` which will be summed with all elements of our list. Overriden occurrence uses recursive call first to compute sum of `tail` (it will return `X`) and invoke method `plus` adding `head` element to the sum. Notice that in this example class there is no any occurrence of classes `Nat`, `Zero` or `Succ` – we were able to express `sum` operation on list using only abstract `plus` which we defined for summables.

You can consider to make a `List` class subtype of `Summable`.

1. What is the meaning of `plus` regarding to lists?
2. How exactly would base class signature would look like?

Let's review erasure of lists implementation.

```
class List extends Object {
  List() { super(); }
  Summable sum(Summable zero_) { return zero_; }
}

class Nil extends List {
  Nil() { super(); }
}

class Cons extends List {
  Summable head;
  List tail;
  Cons(Summable head, List tail) {
    super(); this.head = head; this.tail = tail;
  }
  Summable sum(Summable zero_) {
    return (Summable)(this.tail.sum(zero_).plus(this.head));
  }
}
```

Again, all type parameters were removed and occurrences of all type variables were replaced with `Summable`. Erasure is optimized in that way that it doesn't insert casts, if they are not necessary – see implementations of `sum` method and references to `zero` argument which are not casted. The only cast we need to insert is around invocation of `plus` method from `Summable`, which still returns `Object`.

Having the context of `Nat` and `List` classes, let's consider such expression:

```
new Cons<Nat>(
  new Succ(new Succ(new Succ(new Zero()))),
  new Cons<Nat>(
    new Succ(new Succ(new Zero())),
    new Nil<Nat>()
  )
).sum(new Zero())
```

...and its erased version:

```
(Nat) new Cons(
  new Succ(new Succ(new Succ(new Zero()))),
  new Cons(
    new Succ(new Succ(new Zero())),
    new Nil()
  )
).sum(new Zero())
```

Recognize the trick? We constructed list of 2 natural numbers (3 and 2) by instantiating `Cons`es with type argument `Nat`, which were removed during erasure. Method `sum` returns `Summable`, but cast to `Nat` were inserted to ensure that both expressions have the same types in corresponding type checkers (both types to `Nat`).

We can now evaluate erased expression using *FJ* evaluator:

```
new Succ(new Succ(new Succ(new Succ(new Succ(new Zero())))))
```

As a result, we got encoding of number 5 which is sum of list elements (3 and 2) with explicit 0 passed to sum.

Example 3 functions as objects

So far we have seen rather simple examples. Now let's try to encode something more advanced.

We want to encode interface for unary functions which takes single argument of type X and returns value of type Y.

```
class UnaryFunc<X extends Object, Y extends Object> extends Object {
  Y ignored;
  UnaryFunc(Y ignored) { super(); this.ignored = ignored; }
  Y apply(X arg) {
    return this.ignored;
  }
}
```

We want to represent simple functions as instances of UnaryFunc class with single method apply for computing function value for given argument. Again, due to lack of interfaces, we have to provide trivial implementation for apply. Since we don't require Y as an argument for method, the trick is to create member of the same type as function's result type and return it in our trivial implementation.

Let's encode simple function for natural numbers, $f(n) = 2 * n + 1$.

```
class TwicePlus1 extends UnaryFunc<Nat, Nat> {
  TwicePlus1(Nat ignored) { super(ignored); }
  Succ apply(Nat n) {
    return n.plus(n).succ();
  }
}
```

Class TwicePlus1 represents that function by replacing multiplication by 2 with addition of arguments and incrementation by calling succ. Notice that since for every natural argument, result of such a function will be positive number, we can encode it within type system by declaring result as Succ type, while still passing Nat as second type argument to UnaryFunc. This is demonstration of aforementioned *covariant method overriding* in *FGJ* – we can declare result type of overriden method as a subtype of result of method declared in super class, even if this type was a type variable – we can now see that subtyping takes care of resolving type variables and actual type arguments passed. That is the reason why we need contexts Δ.

Let's review erasure of classes UnaryFunc and TwicePlus1.

```
class UnaryFunc extends Object {
  Object ignored;
  UnaryFunc(Object ignored) { super(); this.ignored = ignored; }
  Object apply(Object arg_) {
    return this.ignored;
  }
}

class TwicePlus1 extends UnaryFunc {
  TwicePlus1(Object ignored) { super(ignored); }
  Object apply(Object n_) {
    return (Nat)((Nat)(n_).plus((Nat)n_)).succ();
  }
}
```

The same as before, generic types were removed from our classes and replaced with their bounds – `Objects`. Covariant method overriding is not present in *FJ*, so erasure had to ensure that types in methods signatures in both classes are identical. Proper casts were inserted in overriden method `apply`:

- two casts around reference to the variable `n_` – to recapture its type, which was `Nat` in example with generic types
- cast to `Nat` around invocation of method `plus`, as well in previous examples.

Combining it together

Let's extend our `List` class to support mapping its elements with unary functions.

```
class List<X extends Summable<X>> extends Object {
  ...
  <Y extends Summable<Y>> List<Y> map(UnaryFunc<X, Y> f) {
    return new Nil<Y>();
  }
}

class Cons<X extends Summable<X>> extends List<X> {
  ...
  <Y extends Summable<Y>> List<Y> map(UnaryFunc<X, Y> f) {
    return new Cons<Y>(f.apply(this.head), this.tail.map(f));
  }
}
```

In base class we added method `map` parameterized with type parameter `Y` which takes unary function and simply constructs empty list of summables `Y`. In `Cons` we return new list with function `f` applied to the `head` element and `tail` mapped by `f`.

How the erasure of the added methods looks like?

```
class List extends Object {
  ...
  List map(UnaryFunc f_) { return new Nil(); }
}

class Cons extends List {
  ...
  List map(UnaryFunc f_) {
    return new Cons(
      (Summable)(f_.apply(this.head)),
      this.tail.map(f_)
    );
  }
}
```

`UnaryFunc` in `map` argument occurs in erased version. Then cast to `Summable` was inserted around `apply` invocation.

Finally, let's construct example program which uses all classes we defined so far.

```
new Cons<Nat>(
  new Succ(new Zero()),
  new Cons<Nat>(
    new Succ(new Succ(new Zero())),
    new Nil<Nat>()
  )
).map<Nat>(new TwicePlus1(new Zero()))
 .sum(new Zero())
```

We construct list `[1,2]`, map it by function `TwicePlus1` and sum all elements of resulting list with 0.

Erased version contains only topmost cast to `Nat` (remember, `sum` result type was `Summable`, but we have concrete subclass here).

```
(Nat)(
  new Cons(
    new Succ(new Zero()),
    new Cons(
      new Succ(new Succ(new Zero())),
      new Nil()
    )
  ).map(new TwicePlus1(new Zero()))
   .sum(new Zero())
)
```

Erased program evaluates to encoding of number `8`, as we expected.

```
new Succ(new Succ(new Succ(new Succ(new Succ(new Succ(new Succ(new Zero())))))))
```

4 Erasure properties

We have seen type erasure in action on programming language, which although simplified to bare minimum, is able to encode, typecheck and evaluate quite advanced examples. We reviewed erasure of all examples and saw types of some of them and they corresponded to types found by generic typechecker. Moreover, erased programs behaved exactly as we expected when we were defining their generic version. Is it matter of convenient examples, or is it kind of general property?

Authors of [1] come with an answer, stating several theorems, which most important are formulated below.

Property 1 (Erasure preserves typing) *For all well-typed* FGJ *class tables, they are well-typed after erasing under* FJ *typing rules.*

This property ensures us that *FGJ* is fully compatible superset of *FJ* regarding to typechecking.

Property 2 (Erasure preserves execution results) *If well-typed* FGJ *program evaluates to some value w in type-passing semantics, then erased program evaluates to erasure of value w in* FJ *evaluator.*

Both theorems are proved in [1]. There are some technical difficulties in proving second theorem, connected with insertion of special synthetic casts during erasure. Finally, behaviour of the program after erasure is equivalent modulo evaluation of synthetic casts.

5 Conclusion

We have discussed one of possible implementation of generic types – *type erasure*. There are several known problems in languages or development platforms built on top of idea of erasing generic types, amongst which the most popular is *Java Virtual Machine*. Deep understanding of pure idea and ability to review although simplified, yet working implementation will allow you to better understand consequences of the limitations and their real roots.

References

[1] A. Igarashi, B. C. Pierce, P. Wadler *Featherweight Java: A Minimal Core Calculus for Java and GJ* 2002.

[2] P. Krzemiński *Slides from "Fundamentals of Object Oriented Languages" seminary* 2014 `http://www.ii.uni.wroc.pl/~dabi/courses/PJZ014/pkrzeminski/fj.pdf`