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 omitting 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. We define two plain classes – `A` and `B`, and class `Pair` containing members `fst` and `snd` and method `setfst` that returns new instance of a pair with modified first element.

```
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 the 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 the full Java
- we always write constructor which initializes all fields defined in that class and calls `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 referred 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

FJ programs consist of a class table and an expression to be evaluated, strictly corresponding to static `main` method in executable Java classes. We intuitively expect that in the context of previously defined classes `A`, `B` and `Pair`, 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 the types of fields `fst` and `snd`, accordingly. Notice that in definition of method `setfst` we introduce another type parameter `Z`, which allows to return an instance of a pair with not only value of type `X` modified, but allowing also to change the first element's type!

```
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 the *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 the 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 the 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 classes, 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 the following case classes extending it:

```
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 overridden 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.

```
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 the `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 a list of classes and returns a 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 the *FJ* paper, which are implemented in `fj.Types`.

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

Main type-checking function is `fj.Types.exprType`. It finds concrete type of expression in given typing context Γ or indicates that the expression is incorrectly-typed. Context Γ contains information about actual types of the available variables and is represented as `Map[VarName, SimpleType]`. There is also an auxiliary function `fj.Types.progType` that type-checks a whole program, ensuring that all the classes, fields, methods are well-typed according to the typing rules, and returns type of the 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 the expression evaluator, some evaluation strategy has to be chosen. This implementation is realized with *call by value* semantics, which corresponds to that known from the full Java, where the 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 a configuration is called *stuck* and the evaluation process cannot be continued. In this implementation we don't bother too much about error handling in the interpreter. When some errored configuration is detected, simply `RuntimeException` is thrown with an appropriate error message, forgetting about the result computed so far.

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

3.1.5 Running examples

Let's consider context of two previous code snippets. Below we present how the type-checker and program evaluator could be launched.

```
println(program.main) // prints: new Pair(new A(), new B()).setfst(new B())
fj.Types.programType(program) // Some(Pair) - type of the main expression
val result = fj.Eval.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 interesting program in our language. In the *FJ* we don't have primitive types, especially numbers. But there is a way to encode the 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()); }
}
```

Following the definition, 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 a 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 the base class `Nat` as wrapping the actual number `this` into object of `Succ` class.

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

This example can be found at `fj.examples.Numbers` and launched by typing:

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

3.2 FGJ module

FGJ-related code is contained in `src/main/scala/fgj` directory. There is syntax for *FGJ* programs defined in `AST.scala` and extended type checker in `Types.scala`. Unlike the *FJ* module, there is no program evaluator[4] available in this implementation.

3.2.1 Types

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

[4]Such a direct evaluator would have to use type passing implementation of generics. Actually, extending the *FJ* evaluator to maintain additional environment for the type variables and their actual denotations is rather straightforward and might be considered as an exercise for the reader. The operational semantics for *FGJ* is defined in [1].

```
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 the type variables (again, internally just strings). The types have two possible forms: *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 the example `Pair` class. 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 definition of the class types, which can be parameterized with arbitrary types. Classes are parameterized by a list of *bounded parameters*. Notice change in the `baseClass` signature which is not only a name reference any more, but became the class type parameterizable by the type variables, like in the example below – where super-class type `Collection` is parameterized with a type variable `X`.

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

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

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

3.2.3 Expressions

AST for the expressions is mostly untouched. Just like in the *FJ*, we have 5 different 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
```

All the occurrences of the `TypeName` in the classes `New` and `Cast` were replaced by a `ClassType`. It is possible to create an object `new Pair<X,Y>(...)`, but it's illegal to construct such an expression `new X()`, in the context where `X` and `Y` are the type variables. Similar restriction concerns the casting target type too. In the class `Invoke` there is additional parameter `typeArgs` which encodes actual type arguments of the method invocation.

3.2.4 Type checker

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

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

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

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

3.3 Type erasure

Implementation of the type erasure rules is located at `src/main/scala/erasure` directory.

3.3.1 Overview

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

- FGJ types to FJ types – `erasure.Erasure.eraseType`
- FGJ expressions to FJ expressions – `erasure.Erasure.eraseExpr`
- FGJ classes to FJ classes – `erasure.Erasure.eraseClass`

And finally, auxiliary function that merges results and translates a whole *FGJ* program to an *FJ* program is `erasure.Erasure.eraseProgram`.

3.3.2 Examples

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

Let's use a function `erasure.Erasure.eraseClass` to generate the 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 has the erasure changed here?

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

Now, let's construct a 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 the erasure it looks almost the same.

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

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

Example 2 summable lists

Let's review another example – lists that can contain some summable elements and 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 an empty list
- `Cons` – list constructor which holds single element `head` of type `X` and rest of the list – `tail` of type `List<X>`

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

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

A method `sum` takes the parameter `zero` which is summed into all elements of our list. Overridden occurrence uses recursive call first to compute sum of `tail` (it will return `X`) and then invokes method `plus` adding `head` element to the final result. Notice that in this class definition there is no single occurrence of classes `Nat`, `Zero` or `Succ` – we were able to express `sum` operation on list using only the abstract `plus` method that we defined for summables.

Let's review erasure of such a summable list 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 the type parameters were removed and all the occurrences of type variables were replaced with their bounds – the `Summable` type. The erasure procedure is optimized in that way that it doesn't insert the casts, if they are not absolutely necessary – see implementations of `sum` method and references to `zero` argument which are not cast. The only cast we need to insert is placed around the invocation of the `plus` method from `Summable`, which still returns an `Object`.

Having the context of the `Nat` and `List` classes, let's consider such an 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())
```

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

Let's evaluate erased expression using the *FJ* evaluator:

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

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

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 an interface for unary functions which takes a single argument of type `X` and returns a 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 the `UnaryFunc` class with single method `apply` for computing function value for the 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 a method, the trick is to create a member of the same type as the function's result type and return it in our trivial implementation.

Let's encode a 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();
  }
}
```

The class `TwicePlus1` represents that function by replacing multiplication by 2 with addition of arguments and incrementation by calling a `succ`. Notice that since for every natural argument, result of such a function will be positive number – we can encode that within the type system by declaring result as a `Succ` type, while still passing `Nat` as a second type argument to the `UnaryFunc`. This is demonstration of aforementioned *covariant method overriding* in the *FGJ* – we can declare the result type of overridden method as a subtype of the result type of method declared in super class, even if this type was a type variable. It's the subtyping that takes care of resolving the type variables and the actual type arguments passed. That is the reason why we needed the 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 the classes and replaced with their bounds – `Objects`. Covariant method overriding is not present in the *FJ*, so erasure had to ensure that types in methods signatures in both classes are identical. Proper casts were inserted in overridden method `apply`:

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

Combining it together Let's extend the `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 the base class we added method `map`, parameterized with the type parameter `Y` that takes unary function and simply constructs an empty list of summables `Y`. In `Cons` we return a 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 the `map`'s argument occurs in erased version. Then cast to `Summable` was inserted around `apply` invocation in the `Cons` class.

Finally, let's construct an example program which uses all the classes 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 the encoding of a 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 the type erasure in action on programming language, which although simplified to bare minimum, is able to encode, type-check 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 type-checker. Moreover, erased programs behaved exactly as we expected when we were defining their generic version. Is it a matter of convenient examples, or is it a kind of general property? Authors of [1] come with an answer, stating several theorems.

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 the *FGJ* is fully compatible superset of *FJ* regarding type-checking.

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

Both theorems are proved in [1]. There are some technical difficulties in proving the second theorem, connected with an 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 programming languages and development platforms built on top of the idea of erasing generic types, amongst which the most popular is *Java Virtual Machine*. Deep understanding of the pure idea and ability to review although simplified, yet working implementation will allow the reader 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*. ACM Transactions on Programming Languages and Systems (TOPLAS), 23(3), May 2001.

[2] P. Krzemiński, Slides from *Fundamentals of Object Oriented Languages* seminar lead by D. Biernacki. Institute of Computer Science, University of Wrocław, 2014. Available at `http://www.ii.uni.wroc.pl/~dabi/courses/PJZ014/pkrzeminski/fj.pdf`.