CSC547: Type Systems for OO Languages

Featherweight Java

Igarashi, Pierce and Wadler, Proc. OOPSLA 1999

Introduction

Featherweight Java (FJ)

Featherweight Generic Java (FGJ)

Translating FGJ into FJ
Introduction

Java!

Featherweight Java is a ‘core Java’ to look at language features.

FJ = Java - lots of features.

FJ = Non-imperative object calculus + classes + casts

Featherweight Generic Java adds genericity (a.k.a. templates done properly).

FGJ = FJ + F-bounded polymorphism.

Why bother?
Introduction

Main results:

Subject reduction for FJ and FGJ ‘Well-typed programs don’t get stuck’.

Soundness of a translation from FGJ to FJ.

This validates part of the Generic Java compiler.

Why translate Generic Java to Java? Why not just compile directly?
Featherweight Java

A sample Featherweight Java program:

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);
    }
}
class A extends Object { A() { super (); } }
class B extends Object { B() { super (); } }

Looks like a Java program!

Some restrictions: no field update, constructors can only initialize fields, ...

What does a linked list class look like?
Featherweight Java

Featherweight Java has two semantics:

Dynamic semantics $e \rightarrow e'$.

Static semantics $\Gamma \vdash e \in C$

Subject reduction says that these semantics agree.

For example:

$$\text{new Pair (new A(), new B()).fst} \rightarrow \text{new A()}$$

and:

$$\vdash \text{new Pair (new A(), new B()).fst} \in A$$

What about:

$$\text{new Cons (new A(), new Cons (new B(), new Nil ())).tail.hd}$$

Gory details in Figure 1 of the full paper (the conference version doesn’t include the rule names!).
Featherweight Java

One technical detail *stupid casts*:

\[(T\text{-SCast})\]
If \(\Gamma \vdash e \in D\)
and \(C \nRightarrow D\)
and \(D \nRightarrow C\)
and *we give a stupid warning*
then \(\Gamma \vdash (C)e \in C\)

Why are these needed?
Featherweight Java

We get subject reduction for FJ...

Theorem (Subject reduction for FJ)

If $\Gamma \vdash e \in C$ and $e \rightarrow e'$ then $\Gamma \vdash e' \in C'$ for some $C' <: C$.

We can also show:

- Well-typed programs produce no method-access or field-access errors.
- Well-typed programs which use only upcasts produce no cast errors.

Hooray.
Featherweight Generic Java

A sample Featherweight Generic Java program:

// template class:
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;
    }
    // template method:
    <Z extends Object>
    Pair<Z,Y> setfst (Z newfst) {
        return new Pair<Z,Y> (newfst, this.snd);
    }
}

What is the type of:

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

Now what does a linked list class look like?
Featherweight Generic Java

GJ supports F-bounded polymorphism, for example:

class Ordered <X> {  
    bool lessThan (X x) { return this.lessThan (x); } // infinite loop!  
}  
class Integer extends Ordered <Integer> {  
    int contents;  
    Integer (int contents) { this.contents = contents; }  
    bool lessThan (Integer x) { return this.contents < x.contents; }  
}  
class SortedTree <X extends Ordered <X>> {  
    ... can use x.lessThan (y) if x : X and y : X ...  
}  

All generic classes are invariant (no variance annotations)!
Featherweight Generic Java

A gotcha...

We have to be careful about generic casts, for example in Generic Java:

```java
Object foo = new List<A>();
List<B> bar = (List<B>)foo;
```

this cast is not allowed, but this one is:

```java
List<A> foo = new List<A>();
ConsList<A> bar = (ConsList<A>)foo;
```

(assuming ConsList<A> extends List<A>).

Why not?
Featherweight Generic Java

Gory details in Figure 3 of the full paper.

Same results as before...

**Theorem (Subject reduction for FGJ)**

If $\Gamma \vdash e \in T$ and $e \rightarrow e'$ then $\Gamma \vdash e' \in T'$ for some $T' <: T$. 
Compiling FGJ to FJ

We can compile FGJ to FJ by:

- Erasing all the generic type parameters.
- Adding *synthetic* casts to recover type information where needed.

For example, Pair `<X,Y>` was:

```java
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);
    }
}
```

To translate this to FJ, we erase all the type parameters and replace them by `Object`. 
Compiling FGJ to FJ

What is the translation of:

```java
new Pair<A,B>(
    new A(),
    new B()
).fst
```

What about:

```java
new Pair<A,B>(
    new A(),
    new Pair<B,C>(
        new B(),
        new C()
    ).fst
).snd
```
Compiling FGJ to FJ

We’d like:

If $e \rightarrow e'$ in FGJ
then $|e| \rightarrow^* |e'|$

(where $|e|$ is the translation of FGJ into FJ).

Unfortunately it’s not true due to casts!

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

(do the snd before the fst and oh dear...)
Compiling FGJ to FJ

Write $e \Rightarrow e'$ if we can get from $e$ to $e'$ by changing synthetic casts.

The result is:

If $e \rightarrow e'$ in FGJ
then $|e| \rightarrow^* d \leftarrow |e'|$ in FJ

which proves that a fragment of the GJ compiler is correct!
Next week

Last formal lecture: *A Concurrent Object Calculus* by Gordon and Hankin.