IMPROVING THE JAVA TYPE SYSTEM

João Loff - 56960
Alexandre Almeida - 64712
Tiago Aguiar - 64870

BEHAVIORS

> *"CtBehavior represents a method, a constructor, or a static constructor"*

```
for (CtBehavior behavior : clazz.getDeclaredBehaviors()){

    behavior.instrument(new ExprEditor() { ... });

    if(behavior instanceof CtMethod) { ...}

    if(behavior instanceof CtConstructor) { ... }
}
```

180 LOC

FIELD INITIALIZATION

We start with:

```
@Assertion("true")
int foo;

foo++;
```

We end up with:

```
static java.util.HashSet f$writes = new java.util.HashSet();

@Assertion("true")
int foo;

if(!f$writes.contains(foo_hash_code)) //...
foo++
f$writes.add(foo_hash_code);
if(!(true)) //...
```

FIELD INITIALIZATION

How to we solve this?

```
@Assertion("true")
int foo;

@Assertion("bar>foo")
int bar;

bar=1;
foo++;
```

DOUBLE CODE INJECTION

FIELD INITIALIZATION

The first injection

```
if(!f$writes.contains(bar_hash_code)) //...
bar=1;
f$writes.add(bar_hash_code);
if(!(bar>foo)) //...
```

The second injection

```
if(!f$writes.contains(bar_hash_code)) //...
bar=1;
f$writes.add(bar_hash_code);
if(!f$writes.contains(bar_hash_code)) //...
if(!f$writes.contains(foo_hash_code)) //...
if(!(bar>foo)) //...
f$writes.add(bar_hash_code);
if(!(bar>foo)) //...
```

METHOD ASSERTION

This **shouldn't** return NPE!

```
Object o = new Object();

public Object getAndClear() {
    return this.o = null;
}

@Assertion("getAndClear() != null")
public void m() {
    this.o.toString();
}
```

METHOD WRAPPING

METHOD ASSERTION

We end up with something **kinda like** this:

```
public void m() {
    // temp "is a" bytecode var
    temp = m$wrapped();
    // now we check assertions
    if(!(getAndClear() != null)) //...
    // and now we return!
    return temp;
}

@Assertion("getAndClear() != null")
public void m$wrapped() {
    this.o.toString();
}
```

A "new" method that returns the result of the original one

METHOD ASSERTIONS

Inherited Method Assertions

```
class Base {
    @Assertion("false")
    public int fooBar(int x) { ... }
}

public class Derived extends Base {
    @Override
    @Assertion("true")
    public int fooBar(int x) { ... }
}
```

We **recursively inject code** from parents assertions

```
public int fooBar(int x) {
    temp = fooBar$wrapped(x);
    if(!(false)) //...
    if(!(true)) //...
    return temp;
}
```

CONSTRUCTOR ASSERTIONS

Remember Behaviors?

```
if((behavior instanceof CtConstructor) && hasAssertion(behavior)){
    CtConstructor constructor = (CtConstructor) behavior;
    constructor.insertBeforeBody(
        "if(!(" + getAssertionValues(constructor) + ")) //... "
    );
}
```

That was easy.

ASSERTIONS ON METHOD ENTRY

We can't break original functionality!

```
Object o = new Object();

public Object getAndClear() {
    return this.o = null;
}

@Assertion(
    value="getAndClear() != null",
    entry="getAndClear() == null"
)
public void m() {
    this.o.toString();
}
```

Previous example, now with entry assertion

ASSERTIONS ON METHOD ENTRY

We inject code at begining of method and at the exit.

```
public void m() {
    // entry assertion
    if(!(getAndClear() == null)) //...
    temp = m$wrapped();
    // value assertion
    if(!(getAndClear() != null)) //...
    return temp;
}
```

It **should** return NPE!

```
## As you can see (getAndClear() == null) expression returned true

[java] Exception in thread "main" java.lang.NullPointerException
[java]     at ist.meic.pa.tests.Base.m$wrapped(Unknown Source)
[java]     at ist.meic.pa.tests.Base.m(Unknown Source)
[java]     ...
[java] The assertion getAndClear() != null is false
```

QUESTIONS?

Notas Apresentação PA

Slide 1:
Focamo-nos mais em mostrar a nossa solução, e não tanto o código que gerou a nossa solução.
De referir também que o código apresentado não é exactamente o mesmo que está no projecto. Está adaptado para ser mostrado em slides.

Slide 2:
Usando behaviors permitiu ter um código mais abstracto para as funcionalidades comuns - percorrer todos os fields - mas permitiu depois especializar comportamento para casos especiais
Parte do comportamento do getDeclaredBehaviors é a actualização dinâmica da loaded ctClass
Esta escolha resultou num codigo bastante compacto que na sua totalidade conta com 180 linhas de código.

Slide 3:
Em termos de assertions em fields, o maior desafio foi a verificação da inicialização.
Resolvemo-la em três simples passos.
Primeiro injectamos um HashSet que irá guardar os fields nos quais já foram feitos writes
Podemos verificar aqui que ao executar um write, introduzimos um add no HashSet antes de verificar a expressão
Ao fazer um read, é primeiro injectado código para verificação se o field já foi escrito anteriormente

Slide 4:
A unica situação que ficou por resolver numa primeira fase foi a verificação de fields com assertions em expressões de outros field assertions.
Resolvemos esta questão injectando código por uma segunda vez sobre código já injectado o que nos permitiu verificar a inicialização dos fields.

Slide 5:
Na primeira injecção apenas estamos a verificar se a asserção bar>foo, mas não estamos a verificar se foo foi inicializado.
Na segunda injecção é verificado a asserção do bar de novo, mas principalmente a asserção do foo é verificada

Slide 6:
Temos de inserir código exactamente "antes" do return, mantendo o valor dos parâmetros iniciais. Também em relação aos argumentos, não esquecer que argumentos ($1, $2 ...) são actualizados automaticamente, logo temos de "preservar" esses valores. O wrapping to método abrange ambas as situações.

Slide 7:
É óbvio que não é feito assim. No entanto, e como bytecode é mais dificil de perceber que java, achamos que esta representação simplificada é melhor.

Slide 8:
Fizemos questão de que as assertions mais próxima da chamada, sejam verificadas primeiro. Também cobrimos situações onde apenas a Base tem a assertion.

Slide 9:
insertBeforeBody insere código no inicio do corpo do construtor.

Slide 10:
Esta solução permite manter a funcionalidade original, construindo sobre essa uma nova funcionalidade para as entradas dos métodos. Permite certas operações que de outra maneira não seria possivel, como seja executar expressões diferentes para inicio e fim.

Slide 11:
A maneira mais fácil de provar a extensão, é mesmo reduzir ao absurdo, executando uma situação completamente errada, e esperar daí uma excepção, tal como o fazemos aqui.