Invariant Specification and Multi-Staging using Java Annotations

Corky Cartwright
Mathias Ricken
Walid Taha

Java Annotations

- **Attach meta-data to program constructs**
  → **Data about the program, not data in the program**
- **Formerly often specified as comments**
  → **Can now be checked and processed automatically**
  → **Avoids parsing of source code or strings**
  → **Annotations act as "smart comments"**
- **Annotations are product types ("structs") composed of constants**
  → **primitives (`int`, `double`, etc.)**
  → **enums**
  → **strings**
  → **class constants (`Integer.class`)**
  → **other annotations**
  → **arrays of the above**

```
@interface OnlyRunByThread {              // annotation definition
    String value;                         // string member
}
@interface NonNull { }                    // "marker" annotation def.

@OnlyRunByThread("main")                  // class annotation
class MyClass {
  @NonNull Object field;                  // field annotation
  MyClass(@NonNull Object param) {        // parameter annotation
    field = param;
  }
  @NonNull Object method() {              // method annotation
    @NonNull Object localVar = field;     // local variable annotation
    return localVar;
  }
}
```

Subtyping

- **Annotations in Java do not have a common supertype**
  → **Cannot define an annotation that can contain ANY other annotation**
- **Added subtyping for annotations to Java**
  → **Minimal changes to compiler, no changes to class file format**
  → **Minor changes to reflection API to support additional features**
- **Integrates well with existing code**
  → **Improves `@DefaultQualifier` annotation, which currently uses a string**

```
// former way to specify more than one default qualifier
@interface DefaultQualifier { String value; }
@interface DefaultQualifiers { DefaultQualifier[] value; }

@interface NonNull  { }
@interface Interned { }

@DefaultQualifiers({@DefaultQualifier("NonNull"),    // use of strings!
                    @DefaultQualifier("Interned")})
class MyClass { ... }
```

```
// specifying more than one default qualifier with subtyping
@interface Annotation { }
@interface DefaultQualifier { Annotation[] value; }

@interface NonNull  extends Annotation { } // subtyping
@interface Interned extends Annotation { } // subtyping

@DefaultQualifier({@NonNull, @Interned})
class MyClass { ... }
```

Additional Targets

- **Annotations in Java cannot be attached to statements or expressions**
- **Allow additional targets for annotations**
  → **block statements**
  → **parenthetical expressions**

```
@Contained {                              // block annotation
  // block of code that does not spawn async. tasks ("contained")
}
int i = -5;
int j = @AlwaysPositive (i*i);            // paren. expression annotation
```

Invariant Specification

- **Program invariants can be encoded as annotations and checked automatically**
  → **Similar to `assert` statements, but inherited into subclasses**
  → **Generates log file instead of terminating program**
  → **Simple generation of invariant index using Javadoc tool**

```
interface TableModel {
    // invariant: must be called from within event thread
    @OnlyEventThread void setValueAt(...);
}

class MyTableModel implements TableModel {
    void setValueAt(...) { /* invariant automatically inherited */ }
}

// from outside event thread...
TableModel m = new MyTableModel(...);
m.setValueAt(...); // invariant violation, generates log entry
```

Multi-Staging

- **Multi-stage programming (MSP) is a paradigm for developing generic software witout paying a runtime penalty for this generality**
  → **"Staging" moves computations into a code generation step before runtime**
  → **Genericly written code (e.g. `power`) is optimized for special cases (e.g. `square`)**
- **Use annotations to mark how expressions and statements should be staged**
  → `@Code` **code to be generated (`.<x>.` in MetaOCaml, "brackets")**
  → `@Escape` **code to be spliced together (`.~x` in MetaOCaml)**
  → `@Run` **run generated code (`.!x` in MetaOCaml)**
- **Staging annotations can be ignored to yield unstaged program**

```
// staged power function in Java
@Code double power(@Code double x, int n) {
  if (n==0) return @Code (1.0);
  else return @Code (@Escape (x) * @Escape (power(x, n-1)));
}
double square(double x) { return @Run (power(@Code (x), 2)); }
```

```
let rec power (x, n) = (* staged power function in MetaOCaml *)
    match n with
      0 ->  .<1>.
    | n ->  .<.~x * .~(power (x, n-1))>.;;
let square = .! .<fun x -> .~(power (.<x>., 2))>.;;
```