Creating a Type System For Use With Verification Games

Verification Games is based on the Checker Inference Framework, which is based on the Checker Framework.[1] As a result, type systems for Verification Games are similar to those for the Checker Framework, which is thoroughly documented. Where appropriate, I have added references to the Checker Framework documentation.

This section will describe the development of the non-negative checker, which prevents negative integers from being used inappropriately. I developed it to better understand the framework. The non-negative checker is small and self-contained, yet it provides simple examples of several different types of behaviors that one might want a checker to exhibit. It should be easy to use these examples to create a more complicated checker. A checker created in the fashion described will work for type checking, but additional work will be required to enable type inference, and to translate those problems into a game. Future work could be to write a guide on how to enable this.

Decide on properties for the type system

The creator of the type system must determine what properties she wishes the type system to have, and what the type annotations should be to accomplish this. The type system will extend Java's type system with a set of additional constraints. The additional type information will be expressed as type annotations that the programmer or an automated tool can insert into the source code. Because of the specific game mechanics of our Verification Games, a type system intended for use with Verification Games should have only two type annotations.

The purpose of the non-negative type system is to prevent negative integers from being used where they will cause errors. Specifically, we will want to require `Array` and `List` indices to be provably non-negative.

For the non-negative type system, we will use two type annotations: `@UnknownSign` and `@NonNegative`. `@NonNegative` is a subtype of `@UnknownSign`. This implies that `@NonNegative int` is a subtype of `@UnknownSign int`.

The non-negative checker will have the following properties:

- All normal subtyping properties .
- `int`s passed to `List.get()` must be `@NonNegative`.
- `Array` indices must be `@NonNegative` .
- `@NonNegative int + @NonNegative int` ==> `@NonNegative int`
- `@NonNegative int * @NonNegative int` ==> `@NonNegative int`
- `@NonNegative int / @NonNegative int` ==> `@NonNegative int`
  (division by zero is a separate problem)

The non-negative type system will not account for integer overflow, nor will it work with other numeric types such as `long` or `float`.

---

1 Papi et. al – http://types.cs.washington.edu/checker-framework/

Create the qualifiers for the type system

Follow the directions in the Checker Framework manual[2] to create the qualifiers. This is a simple declarative process that defines some basic properties about the annotations, such as:

- The subtyping relationships
- The default annotation to be applied to variables implicitly
- Which annotation to apply to literals of different types (if different from the default annotation)

The annotations should be defined in the `<checkername>.quals` package, where "quals" stands for qualifiers.

The qualifier definitions for the non-negative checker are below. These are in the `nonnegative.quals` package.

```
@TypeQualifier
@SubtypeOf({})
@Target({ElementType.TYPE_USE, ElementType.TYPE_PARAMETER})
@DefaultQualifierInHierarchy
public @interface UnknownSign { }
```

`@UnknownSign` is a type qualifier, with no supertypes. It can be used wherever types are written, and if a type is not annotated, it is assumed to be an `@UnknownSign`.

```
@TypeQualifier
@SubtypeOf(UnknownSign.class)
@Target({ElementType.TYPE_USE, ElementType.TYPE_PARAMETER})
public @interface NonNegative { }
```

`@NonNegative` is a subtype of `@UnknownSign`.

Create a JDK Stub file

A stub file allows a type system author or user to specify type annotations that a method signature should have without modifying the method itself. In particular, most practical type systems should include a JDK stub file specifying annotations for JDK methods. Stub files are also useful when verifying real-world code that makes heavy use of libraries. The stub file format is specified in the Checker Framework manual.[3]

The non-negative checker includes a stub file. This is how we implement the property that the index passed to `List.get()` must be `@NonNegative`. The relevant part of the stub file is shown below:

2 http://types.cs.washington.edu/checker-framework/current/checkers-manual.html#writing-a-checker
3 http://types.cs.washington.edu/checker-framework/current/checkers-manual.html#stub-format

```
interface List<E> {
    public E get(@NonNegative int index);
}
```

We could enforce more constraints, such as requiring that the index passed to `List.set()` also be `@NonNegative`, but this is sufficient to demonstrate the use of a stub file.

Create a Checker class

A checker class contains boilerplate code expressing basic information about a type system.

It should extend `games.GameChecker`, since we intend to use it in a game. The `GameChecker` provides some behavior common to all game type systems.

It is useful to create fields, of type `AnnotationMirror`, corresponding to the annotations in a type system. An `AnnotationMirror` is the compiler's representation of an annotation.[4] To get an `AnnotationMirror` from the class literal for an annotation, you could write:

```
javacutils.AnnotationUtils.fromClass(processingEnv.getElementUtils(), <annotation class literal>)
```

The class should override the `initChecker()` method and set these fields within it.

An `AnnotatedTypeMirror`[5] is the JSR-308 compiler's representation for a type, including its annotations. It is based on the standard compiler's `TypeMirror`,[6] which represents a type.

Other methods to override:

- `createInferenceVisitor()`
  - Return the visitor for the type checker (see the next section). In the non-negative checker, this returns an instance of `NonNegativeVisitor`.
- `needsAnnotation(AnnotatedTypeMirror)`
  - Used for inference. For type-checking only, return `false`.
- `defaultQualifier()`
  - Return the `AnnotationMirror` for the default qualifier.
- `defaultQualifier(AnnotatedTypeMirror)`
  - Allows for fine-grained control over the qualifier that a type is considered to have if none is written explicitly. For most cases, including the non-negative checker, it is sufficient to return the default qualifier.
- `selfQualifier()`

4 http://docs.oracle.com/javase/7/docs/api/javax/lang/model/element/AnnotationMirror.html
5 http://types.cs.washington.edu/checker-framework/current/api/checkers/types/AnnotatedTypeMirror.html
6 http://docs.oracle.com/javase/7/docs/api/javax/lang/model/type/TypeMirror.html

    - This is currently unused, but it still must be overridden for the checker to compile. Returning the default qualifier is fine.
- `withCombineConstraints()`
    - This is also currently unused. Simply return `false`.

Create a Visitor class

The visitor class contains a little bit more configuration boilerplate, and defines any special requirements that your type system introduces. For example, in the `NonNegativeVisitor`, the requirement that array indices be `@NonNegative` is enforced.

The visitor class should extend `games.GameVisitor`, with the checker class as a type parameter. This will make the visitor an indirect subclass of the Checker Framework's `SourceVisitor`.[7]

Methods:

- Constructor
    - Three arguments: `<your checker class> checker, InferenceChecker ichecker, boolean infer`
    - You must provide these arguments to the superclass constructor. The first makes your checker available to the type-checking code. The second two are for inference, and when instantiating the visitor, you may pass in `null` and `false` for these.
- `createRealTypeFactory()`
    - Return a new instance of your `AnnotatedTypeFactory`, which will be defined below. This allows the type-checking code to make use of your `AnnotatedTypeFactory`.
- Any visitor methods[8] you must override to implement your type system's rules.
    - Any AST node that should have additional constraints enforced upon it should have its visitor method overridden here.
    - For any constraint that you wish to enforce, call the `mainIsNot` method with the following arguments:
        - The `AnnotatedTypeMirror` for which you would like to assert some property.
        - The `AnnotationMirror` representing the qualifier that the type must not have.
        - An error message.
        - The AST `Tree`[9] object corresponding to the first argument.

The `NonNegativeVisitor` overrides `visitArrayAccess` to enforce the constraint that

7 http://types.cs.washington.edu/checker-framework/current/api/checkers/source/SourceVisitor.html
8 http://types.cs.washington.edu/checker-framework/current/api/checkers/source/SourceVisitor.html
9 http://docs.oracle.com/javase/7/docs/jdk/api/javac/tree/com/sun/source/tree/package-summary.html

`Array` indices be `@NonNegative`. The code that enforces this follows:

```
public Void visitArrayAccess(ArrayAccessTree node, Void p) {
    super.visitArrayAccess(node, p);

    ExpressionTree index = node.getIndex();
    AnnotatedTypeMirror type =
        atypeFactory.getAnnotatedType(index);
    mainIsNot(type, realChecker.UNKNOWN_SIGN,
              "unknown.array.index", index);

    return null;
}
```

This method gets the AST tree node for the index expression, then gets its corresponding `AnnotatedTypeMirror` from the `AnnotatedTypeFactory` (which we will define below). Then, it enforces that the index not have the type `@UnkownSign`.

Create an AnnotatedTypeFactory

The `AnnotatedTypeFactory` is what the type-checker uses to find the type of a given AST tree node. Here, you will define any special rules for determining the type of a given tree node. For example, the non-negative checker automatically assigns the `@NonNegative` type to any non-negative integer literals, and also to the addition, multiplication, or division of any two `@NonNegative` integers. These properties are implemented in the `NonNegativeAnnotatedTypeFactory`.

It should subclass `games.GameAnnotatedTypeFactory`

Methods:

- Constructor:
  - Should take an instance of the checker as a parameter, pass it to the superclass constructor, and call `postInit()`
- `createTreeAnnotator()`
  - Should return an instance of a `TreeAnnotator` (see below).

The `TreeAnnotator`[10] is what visits the AST nodes and applies any annotations to them. By overriding `TreeAnnotator` methods, a type system author can implement special rules for what types are assigned to AST nodes. The `AnnotatedTypeFactory` should include a subclass of `TreeAnnotator` as an inner class.

`TreeAnnotator` Methods:

10 http://types.cs.washington.edu/checker-framework/current/api/checkers/types/TreeAnnotator.html

- Constructor:
  - Should call the superclass constructor with the enclosing instance of the `AnnotatedTypeFactory` as an argument.
- Visitor methods[11]
  - Required whenever AST nodes need special logic when determining their type.

For the non-negative checker, we override `visitLiteral` and `visitBinary`.

`visitLiteral` is used to apply the `@NonNegative` annotation to non-negative integer literals:

```
public Void visitLiteral(LiteralTree tree, AnnotatedTypeMirror
type) {
    if (tree.getKind() == INT_LITERAL) {
        if ((int) tree.getValue() >= 0) {
            type.addAnnotation(nnChecker.NON_NEGATIVE);
        } else {
            type.addAnnotation(nnChecker.UNKNOWN_SIGN);
        }
    }
    return super.visitLiteral(tree, type);
}
```

The code inspects the `Tree` for the literal, and if it is an integer literal, looks at its value. If it is non-negative, it assigns it the `@NonNegative` annotation.

In a similar manner, `visitBinary` is used to apply the `@NonNegative` annotation to certain integer arithmetic operations. For example, a `@NonNegative int` plus a `@NonNegative` int is a `@NonNegative int`.

11 http://types.cs.washington.edu/checker-framework/current/api/checkers/types/TreeAnnotator.html#method.summary