Verification Games Type Systems

Nathaniel Mote
Bachelor's Thesis
December 2013

Abstract:

This paper will summarize the process of creating a Verification Games type system and document several existing Verification Games type systems. Verification Games is a project that turns type inference problems into a puzzle game that anyone can play, rather than requiring experts to insert type information manually. I will document the process of creating a type system using examples from one that I created. I will also describe several type systems based on our Trusted type system, and suggest improvements for them.

Introduction

Formal verification can prove the absence of certain classes of bugs in a program. One way to perform verification is to use a type systems with which stronger properties can be obtained than with those that are ordinarily used in programming. An existing program can be verified by adding additional type information to indicate certain properties.

Adding the proper type information is a process that typically requires significant effort on the part of highly skilled, and therefore expensive, programmers. This burden can be eliminated by using type inference, which works well when the program can be proven correct simply by adding a set of annotations.

However, when there is no set of annotations that can lead to successful type checking, type inference fails. This could be because the program truly contains bugs, or it could simply be that the type system is not expressive enough to prove that the program is correct.

In either case, it is still useful to get a partial set of annotations that results in a minimal number of type-checking errors. This would allow part of the system to be verified and for the few failure points to be manually inspected. Ideally, these failure points would indicate the general locations where true problems may occur.

Verification Games reduces the type inference problem to a puzzle game playable by non-experts. The Verification Games team believes that human players will be able to leverage their intuition and produce better incomplete solutions than traditional type inference tools.

When a player produces a solution to the puzzle, his solution can be directly translated back into a set of annotations. If the solution has no conflicts, it means that once the annotations are inserted into the original source code, type checking will succeed. If the solution has conflicts, it means that type checking will fail, but the locations in which it fails should provide insight into what could go wrong in the program at runtime. Further specifics of the game, and how constraints are translated into it, are beyond the scope of this paper.[1]

This paper will focus on the type systems used with Verification Games. First, it will discuss how to create a type system for use with Verification Games, and then it will summarize the trusted type systems – a set of type systems that are based on proving that certain values are trustworthy to be used for some purpose – and suggest improvements for them.

1 Dietl et. al. 2012

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.[2] 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)

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

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

Create the qualifiers for the type system

Follow the directions in the Checker Framework manual[3] 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.[4]

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

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:

```
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.[5] To get an `AnnotationMirror` from the class literal for an annotation, you could write:

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

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

An `AnnotatedTypeMirror`[6] is the JSR-308 compiler's representation for a type, including its annotations. It is based on the standard compiler's `TypeMirror`,[7] 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

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

if none is written explicitly. For most cases, including the non-negative checker, it is sufficient to return the default qualifier.

- `selfQualifier()`
  - 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`.[8]

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[9] 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.

8 http://types.cs.washington.edu/checker-framework/current/api/checkers/source/SourceVisitor.html
9 http://types.cs.washington.edu/checker-framework/current/api/checkers/source/SourceVisitor.html

- The AST `Tree`[10] object corresponding to the first argument.

The `NonNegativeVisitor` overrides `visitArrayAccess` to enforce the constraint that `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`[11] 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

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

what types are assigned to AST nodes. The `AnnotatedTypeFactory` should include a subclass of `TreeAnnotator` as an inner class.

`TreeAnnotator` Methods:

- Constructor:
  - Should call the superclass constructor with the enclosing instance of the `AnnotatedTypeFactory` as an argument.
- Visitor methods[12]
  - 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`.

Trusted Type Systems in Verification Games

One of the goals of the Verification Games project is to protect against a set of security vulnerabilities. One theme that is common in many of them is unintended information exposure or the use of potentially malicious data without first sanitizing it.

The trusted type systems are designed to guard against some of the vulnerabilities listed in the

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

2011 CWE/SANS Top 25 Most Dangerous Software Errors.[13] The trusted type systems are a class of type systems based on two qualifiers: `@Trusted` and `@Untrusted`. Routines can require that their arguments be trusted. Specific type systems have specific meanings for their trusted and untrusted qualifiers. End users can also use the trusted type system directly, and choose what they want the qualifiers to represent.

Trusted

The original trusted type system provides functionality common to all trusted type systems, allowing other type systems to extend it and get its functionality at little cost.

This type system is made up of `@Trusted` and `@Untrusted` annotations, where `@Trusted` is a subtype of `@Untrusted`. It has the ordinary subtyping rules and one additional rule: When the `+` operator is applied to two `@Trusted` types, the result is also a `@Trusted` type. This is intended for use with `String`s, but can also be applied to other types, such as `int`s. Future work could be to either extend it to apply to other arithmetic operators, or to restrict it to apply only to `String`s. In its current state, it is inconsistent, though still entirely usable. All of the trusted type systems implement this behavior, unless otherwise noted. Therefore, trusted type systems that intend for annotations to be used on types other than `String`s also suffer from this problem.

In this type system, types are assumed to be `@Untrusted` unless specified otherwise, except that all literals are `@Trusted` by default. Unless noted below, each trusted type system assumes types to be `@Untrusted` by default, but they have varied rules for what type literals are considered to be.

This type system includes all of the machinery to allow a type inference problem to be translated into a game. This means that any new type system that extends trusted will automatically be able to be gamified. To base a type system on trusted, follow the steps to create a type system above, but instead of having the checker class extend `games.GameChecker`, have it extend `trusted.TrustedChecker`, and set the `UNTRUSTED` and `TRUSTED` `AnnotationMirror` constants in the `setAnnotations()` method. The steps after creating the checker class are optional, and are only necessary if the default behavior provided by the trusted type system is insufficient.

Download

The download checker protects against CWE-494: Download of Code Without Integrity Check.[14] It uses two qualifiers: `@ExternalResource` and `@VerifiedResource`, where `@VerifiedResource` is a subtype of `@ExternalResource`. Routines requiring verified data can annotate their arguments as `@VerifiedResources`. It is up to the user to supply a routine that appropriately sanitizes data, taking an `@ExternalResource` and returning a

13 http://cwe.mitre.org/top25/index.html#Listing
14 http://cwe.mitre.org/top25/index.html#CWE-494

`@VerifiedResource`, if such a routine is necessary for the correctness of the program.

A `@VerifiedResource` could either have been downloaded and passed an integrity check, or it could have originated within the program. Therefore, all literals are considered to be `@VerifiedResources`.

Encoding

The encoding checker protects against CWE-838: Inappropriate Encoding for Output Context.[15] It has two qualifiers: `@UnknownEncoding` and `@AppropriateEncoding`, where `@AppropriateEncoding` is a subtype of `@UnknownEncoding`. The user must determine what an appropriate encoding is. This type system is intended to be used only with Strings and other object types, and is not intended for use with primitive types. `null` literals are given an `@AppropriateEncoding` type.

Encrypted

The encrypted checker protects against CWE-311: Missing Encryption of Sensitive Data.[16] It has two qualifiers: `@Plaintext` and `@Encrypted`, where `@Encrypted` is a subtype of `@Plaintext`. A routine that transmits data over the network or saves it to permanent storage could require that the data it is passed is `@Encrypted`. `null` literals are considered `@Encrypted`, and `String` literals are considered `@Plaintext` by default. This type system is not intended for use with primitive types.

File type

The file type checker protects against CWE-434: Unrestricted Upload of File with Dangerous Type.[17] It was two qualifiers: `@UnknownFileType` and `@SafeFileType`, where `@SafeFileType` is a subtype of `@UnknownFileType`. A web server could require that files received from clients have a `@SafeFileType`. `null` literals are considered to have a `@SafeFileType`.

Hard-coded

The hard-coded checker protects against CWE-798: Use of Hard-coded Credentials.[18] It has two qualifiers: `@MaybeHardCoded` and `@NotHardCoded`, where `@NotHardCoded` is a subtype of `@MaybeHardCoded`. Routines that send credentials to authenticate with a remote server, or routines that perform authentication for remote clients, can require that the credentials are `@NotHardCoded`. This prevents credentials, whether they are for a remote server or used to enable behavior in the program itself, from being hard-coded where they could easily be discovered. All literals except `null` are considered to be `@MaybeHardCoded`.

15 http://cwe.mitre.org/data/definitions/838.html
16 http://cwe.mitre.org/top25/index.html#CWE-311
17 http://cwe.mitre.org/top25/index.html#CWE-434
18 http://cwe.mitre.org/top25/index.html#CWE-798

This type system reuses less of the trusted system's behavior, since it implements a subtly different rule for concatenation with the `+` operator. Instead of requiring that both operands be `@NotHardCoded` (the `@Trusted`-like qualifier) for the result to be `@NotHardCoded`, it is sufficient for only one of them to be `@NotHardCoded`.

Internal

The internal checker protects against CWE-209: Information Exposure Through an Error Message.[19] It has two qualifiers: `@Internal` and `@Public`, where `@Public` is a subtype of `@Internal`. In this type system, `@Public` is actually the `@Trusted` qualifier. It indicates data that is appropriate to be exposed to the end user. `@Types` are `@Internal` by default, and routines that expose information to the end user should require their arguments to be `@Public`. `null` literals are considered to be `@Public`.

This type system includes a JDK stub file that requires arguments to the `print, println`, and `printf` methods of `PrintStream` to be `@Public`. These annotations cover the most common places for private information to escape, but additional annotations are necessary to prevent it entirely. For example, a programmer dedicated to circumventing the type system could leak private information with the write method, which takes bytes directly.

OS Trusted

The OS Trusted checker protects against CWE-78: Improper Neutralization of Special Elements used in an OS Command.[20] It has two main qualifiers: `@OsUntrusted` and `@OsTrusted`, where `@OsTrusted` is a subtype of `@OsUntrusted`. All literals are considered `@OsTrusted`.

Unlike other Trusted type systems it contains an additional qualifier, `@PolyOsTrusted`. This is a polymorphic type annotation, and it allows a method to indicate that whatever type it receives, it also returns. A trivial example is the identity function: If it receives an `@OsTrusted` argument, it also returns `@OsTrusted`, but if it receives an `@OsUntrusted` argument, it also returns `@OsUntrusted`. More documentation on polymorphic type annotations is available in the Checker Framework manual.[21]

The included JDK stub file requires that `String`s passed to OS commands such as `exec` be `@OsTrusted`. It is the responsibility of the user to provide a routine that adequately sanitizes `@OsUntrusted` data and returns `@OsTrusted` data, if such a function is necessary to perform.

19 http://cwe.mitre.org/data/definitions/209.html
20 http://cwe.mitre.org/top25/index.html#CWE-78
21 http://types.cs.washington.edu/checker-framework/current/checkers-manual.html#qualifier-polymorphism

Random

The random checker protects against CWE-330: Use of Insufficiently Random Values.[22] It has two qualifiers: `@MaybeRandom` and `@Random`, where `@Random` is a subtype of `@MaybeRandom`. Routines that should use a cryptographically secure source of randomness, such as key generation routines, should require that their source of randomness be Random.

While `@MaybeRandom` is the default qualifier, the included JDK stub file explicitly annotates the methods in the `java.util.Random` class as `@MaybeRandom`, since they do not use a cryptographically secure source of randomness. In contrast, the methods in the `java.security.SecureRandom` class are annotated as returning Random values.

Salt

The salt checker protects against CWE-759: Use of a One-Way Hash without a Salt.[23] It has two qualifiers: `@MaybeHash` and `@OneWayHashWithSalt`, where `@OneWayHashWithSalt` is a subtype of `@MaybeHash`. Routines that write data to a password file can require that their arguments be a `@OneWayHashWithSalt`, rather than a plain-text password or a hashed password that did not use a salt. `null` literals are considered to be `@OneWayHashWithSalts`.

SQL Trusted

The SQL Trusted checker protects against CWE-89: Improper Neutralization of Special Elements used in an SQL Command.[24] It has two qualifiers: `@SqlUntrusted` and `@SqlTrusted`, where `@SqlTrusted` is a subtype of `@SqlUntrusted`. Routines that submit queries to a SQL server should require `@SqlTrusted` arguments. Literals are considered `@SqlTrusted`.

The annotated JDK stub file adds these requirements to the `java.sql` SQL API.

The user of the type system must provide some way to sanitize `@SqlUntrusted` input, if such functionality is necessary.

22 http://cwe.mitre.org/data/definitions/330.html
23 http://cwe.mitre.org/top25/index.html#CWE-759
24 http://cwe.mitre.org/top25/index.html#CWE-89

Conclusion

In this paper, I have documented the process of creating a type checker for use with Verification Games in type checking mode. I have supplemented this with examples from the non-negative type checker, which I created. I have also described the trusted type systems, and suggested improvements to them.

Future Work

Future work could include adding polymorphic annotations for all the trusted type systems. Currently they only exist for the OS Trusted type system. The addition of polymorphic annotations to other trusted type systems would improve their expressiveness.

The semantics of the `+` operator should be considered more thoroughly. The trusted type systems consider two `@Trusted` operands to the `+` operator to be `@Trusted`. For most trusted type systems, this works well: The concatenation of two `@Trusted Strings` should be `@Trusted`. It can also be applied to other types, but it is the only operator that currently has special rules. For consistency, it should either be limited to `Strings` (with which other operators cannot be used), or extended to other operators. For example, applying the `^` (XOR) operator to two `@Random ints` should result in another `@Random int`.

While the process of creating a type checker that can potentially be used with Verification Games has now been documented and is reasonably straightforward, the process of using it for type inference and using it to create games is poorly documented and more difficult. This is partially due to the fact that the project is still in its infancy, but it should undoubtedly be improved and documented.

References

2011 CWE/SANS Top 25 Most Dangerous Software Errors
http://cwe.mitre.org/top25/index.html

“Improving and Extending Verigames”
Stephanie Dietzel
2013

“Practical Pluggable Types for Java”
Matthew M. Papi; Mahmood Ali; Telmo Luis Correa Jr.; Jeff H. Perkins; Michael D. Ernst
ISSTA 2008, Proceedings of the 2008 International Symposium on Software Testing and Analysis

“Verification Games: Making Verification Fun”
Werner Dietl; Stephanie Dietzel; Michael D. Ernst; Nathaniel Mote; Brian Walker; Seth Cooper; Timothy Pavlik; Zoran Popović
Proceedings for FTfJP 2012: The 14th Workshop on Formal Techniques for Java- Like Programs - Co-located with ECOOP 2012 and PLDI 2012, Papers Presented at the Workshop. 2012:42-49.