Checker Framework Inference Development

Checker Framework-Inference is a framework to infer annotations for programs checked by Checker Framework checkers.

Checker Framework-Inference has multiple modes. For an input program, it can:

- Produce a JAIF of inferred annotations to be inserted into the program.
- Use inferred annotations to immediately type-check the program.
- Output a file of serialized constraints to allow out-of-band solvers to infer annotations for the program.

Checker Framework Type Checking

A checker's visitor enforces the type-checking rules of the checker. As an example, most visitors enforce that the annotations on the type of the *rhs* of an assignment are a subtype of the annotations on the type of the *lhs.* The code

```
a = b;
```

be checked by most visitor's `visitAssignment` method. From `visitAssignment` an eventual call to QualifierHierarchy.`isSubtype` ensures that the annotations on the type of `b` are a subtype of the annotations on the type of `a`. If the check fails, a type-checking warning is emitted to the user.

During type-checking, every annotatable position in the input program has a annotation specified by a combination of manual annotation, implicit annotation rules, defaulting rules, and a dataflow algorithm. For a program to type-check the annotations of the program must satisfy all the applicable rules of the "target" checker performing the type-checking.

Inference

Instead of using default annotations, Checker Framework-Inference (CFI) tries to infer the annotations necessary for the program to type-check. If a program can be type-checked, CFI will always infer annotations such that the resulting program type-checks. Otherwise, CFI tries to infer annotations such that the resulting program will be as close as possible to type-checking (although close to type-checking is not well-defined…).

A program type-checks if every check performed by the visitor passes. During inference, instead of enforcing checks, a CFI visitor generates constraints for every check that would have been performed to ensure that the annotations on the types involved satisfy the requirements of the check. These constraints are formulated as relationships, like subtype or equality, between annotations on the types as well as other annotations from the type system. An AnnoPos object, described in more detail later, is used to represent a specific annotatable position. Inference solves the generated system of constraints to determine a resulting annotation value for every AnnoPos.

As an example, when the CFI visitor visits

```
a = b;
```

the visitor generates a constraint to record that the annotations on `b`'s type must be a subtype of the annotations on `a`'s type. If any of the annotatable positions on `a` or `b`'s type do not yet have annotations, each of the annotatable positions is represented by an AnnoPos object.

One major contribution of CFI is to allow a type system author to create a single visitor that supports both type-checking and constraint generation modes without duplicating code. Assignability checks and subtype checks can be written exactly as they are written in current Checker Framework visitors. CFI will either enforce the check immediately or generate a constraint depending on the mode of the visitor.

However, other types of checks must be written differently. Current Checker Framework visitors are allowed to directly check for the existence of a specific annotation on a type, (e.g. check that a type has the `@NonNull` annotation), with a java == or `.equals()` call. CFI cannot track the use or intent of these calls, so it is unable to generate constraints based on them. CFI visitors must instead use framework-provided helper methods, inference-check-methods. Examples of which include `areComparable`, `areEqual`, `mainIs`, and `mainIsNot`. When CFI is integrated into Checker Framework, existing visitors will have to be converted to use inference-check-methods.

Converting a Checker Framework checker to a Checker Framework Inference checker

Follow these steps to add inference to a Checker Framework checker:

- Code review the Checker Framework checker and ensure it is up to date.
- Copy the checker's code to the `src/` directory in a clone of the Checker Framework Inference repository. For example, the Interning Checker can be found under `src/interning`.
- Modify the checker's package inside each java file to the form used by Checker Framework Inference. For example, package `org.checkerframework.checker.interning` is renamed to `interning`. Fix imports accordingly in each java file to ensure classes from the Checker Framework's version of the checker (if the original checker is shipped with the Checker Framework) are not accidentally imported, as this could lead to subtle bugs. Remove the `package-info.java` file if it is present.
- See `src/interning/InterningChecker.java` for an example of how to modify the Checker class. The steps are: Modify the Checker class to extend `BaseInferrableChecker` instead of `BaseTypeChecker`. Override the `initChecker()` method to initialize `AnnotationMirrors` (that are members of the checker) for commonly used annotations in the typesystem (and call `super.initChecker()`). Override `createVisitor` and `createRealTypeFactory` to return new instances of the visitor and type factory for the checker.
- Modify the Visitor class to override `InferenceVisitor` instead of `BaseTypeVisitor`. Ensure the `InferenceVisitor` takes as a type parameter the Checker class as well as `BaseAnnotatedTypeFactory` instead of the annotated type factory for the typesystem. Modify the Visitor constructor's signature to that of the `InferenceVisitor` constructor (and call the super constructor). See `src/interning/InterningVisitor.java` for an example of how to do this.
- In the Visitor class, look for instances of `checker.report` and the if blocks they are contained in.
    - If the `checker.report` call is expressing an expectation about a type qualifier for a given type, convert the if block and `checker.report` call to one of the utility methods in `InferenceVisitor` that calls `checker.report` while in typechecking mode and generates a constraint while in inference mode. For example, given a condition that looks like:

```
if (!annotatedTypeMirror.hasEffectiveAnnotation(NONNULL))
    checker.report(Result.failure("should.be.nonnull", annotatedTypeMirror),
expressionTree);
```

Replace it with

```
mainIs(annotatedTypeMirror, realChecker.NONNULL, "should.be.nonnull",
expressionTree);
```

This will behave the same way as the original if block while in typechecking mode (with possible cosmetic differences in the output), and will generate a constraint that the type annotation should be `@NonNull` while in inference mode. Using annotations from `realChecker` rather than from the visitor is a convention in Checker Framework Inference. See `src/interning/InterningVisitor.java` for examples of calls to `mainIs`.

- If the `checker.report` call is expressing an expectation about a declaration annotation, such as an annotation on a class, it is often best to leave it unchanged. This is because such annotations cannot be inferred, and there is a general philosophy that if inference completes without errors, typechecking should also complete without errors. Because of this, these errors should also be issued while in inference mode. However, if the situation warrants that these errors only be issued while in typechecking mode, one can use the `infer` property of the visitor to determine whether the checker is running in inference mode and not issue errors in that case, as in:
  `if (!infer) { checker.report(...); }`

- Because defaults are not used in inference mode, defaults specified in type qualifier definitions may require additional constraints to be specified while in inference mode. For example, suppose a type qualifier `@Foo` specifies in its definition that all primitive types will be defaulted to `@Foo`. One or more `mainIs( …, realChecker.FOO, … )` calls need to be added to the visitor when visiting primitive types to indicate that this is the case.
- Lint options are not available in inference mode, so ensure that processing of lint options is enclosed in an `if (!infer) { … }` block.

Execution

This is a summary of the execution flow of CFI when inferring annotations for a program. Components (like AnnotatedTypeFactory or QualifierHierarchy) of the checker thats types are being inferred are prefixed with the term “target.” Further descriptions for the classes involved can be found in the Class Overview section.

CFI is started as a regular Java program by executing the InferenceCli class (Step 1 in the Execution Diagram). InferenceCli creates an instance of InferenceMain to handle the rest of the inference process (Step 2). This InferenceMain instance is made accessible by the rest of CFI through a static method.

InferenceMain invokes the Checker Framework programmatically using the javac api to run the InferenceChecker (Step 3). The Checker Framework creates an instance of the InferenceChecker (Step 4) and executes the InferenceChecker as a normal type-checker. Since javac is running in the same JVM and with the same classloader as InferenceMain, the InferenceChecker can access the static InferenceMain instance.

During its initialization, InferenceChecker uses InferenceMain to get an instance of the InferenceVisitor (Step 5). InferenceMain creates the InferenceVisitor by calling methods on the InferrableChecker’s interface (Step 6). The Checker Framework retrieves the InferenceVisitor from the InferenceChecker (Step 7) and then uses this visitor to type-check the source code (Step 8).

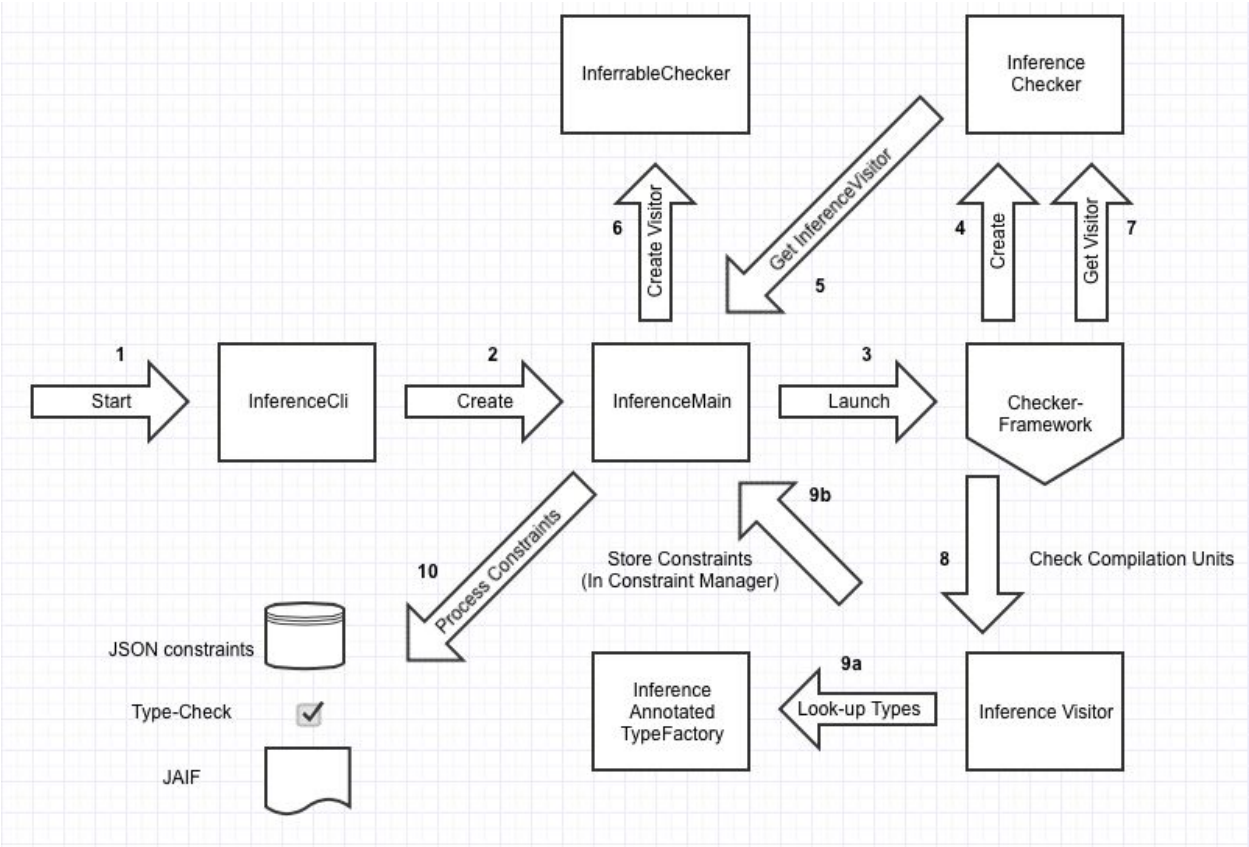
For every compilation unit (source file) in the program, the InferenceVisitor scans the AST and generates constraints where each check would have occurred. InferenceMain creates and holds a reference to a

ConstraintManager instance to store all constraints generated (Step 9a). During constraint generation the InferenceVisitor also looks up types using the InferenceAnnotatedTypeFactory (Step 9b). InferenceAnnotatedTypeFactory returns types where each annotatable position in the type is annotated with either an annotation from the real type system, or an `@AnnoPosId` annotation which corresponds to an AnnoPos object.

After the last compilation unit has been scanned by the visitor, the Checker Framework call completes and control returns to InferenceMain. InferenceMain checks the return code of javac. The Checker Framework will return an error if no source files were specified, if the specified source files did not exist, or if the source files fail to compile. Error codes for other reasons generally result from bugs in CFI; inference only generates constraints, it does not enforce type-checking rules.

If the Checker Framework call does not return an error, CFI will then process the generated constraints. CFI can solve the constraints using and InferenceSolver and generate a JAIF to allow insertion of inferred annotations back into the input program (Step 10). InferenceSolver is an interface that all solvers must implement.

CFI can also serialize the constraints for processing later (by a solver or by Verigames). In the future, CFI might be able to use the inferred annotations for type-checking without first inserting the annotations into the input program.



Execution Diagram

AnnoPos
AnnoPos is a class that represents possible annotatable positions in a program.

During inference, every possible annotatable position is given one of two types of annotations. Annotatable positions are assigned an annotation from the target type system if there is no source code for the declaration of the type (library code). Currently these annotatable positions are assumed to be fixed and are assigned the result of looking up the type with the target AnnotatedTypeFactory which applies stubfile and defaulting rules. Annotations from the target type system are considered “constant” annotations.

For all of the other annotatable positions, each position is represented by a unique AnnoPos object that has a unique `id` field. Besides representing an annotatable position, an AnnoPos holds other fields used by inference. The most important field is an ASTPath `position` which specifies the AnnoPos’ annotatable position in the source code in a format usable by the Annotation File Utilities. This `position` is used at the end of inference to generate a JAIF file that can be used to insert the inferred annotations into the input program.

It would be convenient for an AnnoPos to be an annotation that could be placed in the annotatable position it represents. However AnnoPos cannot be an Annotation class because of Java type restrictions on the field type of Annotations. Instead, annotatable positions are given an `@AnnoPosId` annotation. `@AnnoPosId` annotations have a single integer field, `id`. The `id` of a `@AnnoPosId` on an annotatable position has the same value as the `id` of the AnnoPos that represents that annotatable position.

This example shows variables annotated with `@AnnoPosId` annotations and an assignment of variable `b` to variable `a`.

```
@AnnoPosId(id=1) String a;
@AnnoPosId(id=2) String b;
a = b;
```

A visitor that enforces subtype relationship for assignments would generate a constraint that the AnnoPos with `id=2` must be a subtype of the AnnoPos for with `id=1`. The solution to the system of constraints for a program is a mapping of AnnoPos to annotations from the target type-system.

Some annotatable positions will already have a known value (a value that does not need to be inferred) because they represent annotatable positions that are already annotated in the source code or because there exist qualifier rules that specify an implicit value for the annotatable position. Each of these annotatable positions are still assigned a AnnoPos, but the AnnoPos will have an equality constraint to the constant value from the target type system.

Class Overview
This is an overview of the main classes in CFI. These classes are located in the `checkers.inference` package. See each class’s documentation for more details.

InferenceCli
InferenceCli is the main launcher and entry point of CFI.

InferenceCli also handles command-line arguments and initializes logging. Command line arguments also configure the output mode: producing a jaif, type-checking, or serializing constraints.

InferenceMain
InferenceMain is the central coordinating class for CFI. It executes the InferenceChecker as a Checker Framework checker and processes constraints when the type checker completes.

InferenceMain uses the InferrableChecker interface of the target checker to instantiate components and wire them together. It creates and holds instances to the InferenceVisitor, the InferenceAnnotatedTypeFactory, the InferrableChecker, etc.

InferenceChecker
InferenceChecker is a Checker Framework checker that generates constraints by using an InferenceVisitor.

InferenceMain invokes the Checker Framework using the InferenceChecker as the checker and the input program to infer as the java source-file parameter. When the Checker Framework process asks InferenceChecker for its visitor, InferenceChecker retrieves the InstanceVisitor by calling back to the static instance of InferenceMain.

InferrableChecker
An InferrableChecker is a checker that can be inferred with CFI. An InferrableChecker should also extend from BaseTypeChecker so that the checker can also be run in type-checking mode.

Target type checkers must implement the InferrableChecker interface. InferrableCheckers contains factory methods that CFI uses to instantiate components of the target checker such as the target AnnotatedTypeFactory, the target QualifierHierarchy, and the InferenceVisitor.

In the future the InferrableChecker interface could configure heuristics or defaults for use in solving constraints. For example, the InferrableChecker could specify which type in the type system is preferred.

InferenceVisitor
An InferenceVisitor is a visitor that can either enforce the type-checking rules of a checker or generate constraints based on the type-checking rules.

The target type-system's visitor must extend from InferenceVisitor and must be implemented using inference-check-methods declared in InferenceVisitor. This visitor is used both for type-checking and for inference; there is no separate inference-only visitor.

InferenceVisitors are instantiated in either type-checking or inference modes. In type-checking mode, the visitor is created by the target checker and enforces checks by emitting user errors when a check fails. In inference mode, the visitor generates a constraint for each check instead of enforcing it.

Visitors use an AnnotatedTypeFactory to lookup the AnnotatedTypeMirrors for trees or elements so that the visitor can check relationships between types. During inference, InferenceVisitor uses an InferenceAnnotatedTypeFactory that returns AnnotatedTypeMirrors annotated with constant annotations from the target type-system and `@AnnoPosId` annotations. This allows constraints generated by the visitor to be specified in terms of relationships between constant annotations as well as AnnoPos.

InferenceAnnotatedTypeFactory
InferenceAnnotatedTypeFactory is an AnnotatedTypeFactory that returns AnnotatedTypeMirrors annotated

with either:

- An annotation from the real type system for types where the source code is not available (library types).
- A `@AnnoPosId` for all other positions.

InferenceAnnotatedTypeFactory ensures that annotatable positions without constant values are always given unique `@AnnoPosId` that maps to the AnnoPos object for that position. InferenceAnnotatedTypeFactory uses a SlotManager instance to map an `@AnnoPosId` to its corresponding AnnoPos object.

Currently the only situation in which the InferenceAnnotatedTypeFactory will return a type with constant annotations is when a type is created for library methods and fields. This is mostly for historical reasons. Instead of returning types with constant annotations, we could treat library methods like preannotated code and use an AnnoPos with an equality constraint to the constant annotation it is substituted for. This would allow us to enforce that no type should ever be returned with a constant annotation, giving us a better assurance that we were annotating the program correctly. However, this has not yet been implemented and we do not think this is currently a priority.

InferenceQualifierHiearchy

InferenceQualifierHierarchy is a QualifierHierarchy for the annotations that can appear on types during inference.

AnnotatedTypeMirrors used in the inference system can have `@AnnoPosId` annotations and constant annotations. `@AnnoPosId` and all target type-system annotations are considered siblings of each other and subtypes of the top annotation, `@InferenceTop`. No types should have `@InferenceTop`.

Merges between a `@AnnoPosId` and a constant annotation will result in a new `@AnnoPosId` annotation to represents the least upper bound of the two annotations that were merged. SubtypeConstraints will be generated to ensure the new `@AnnoPosId` is a supertype of the the two input annotations.

InferenceQualifierHierarchy also creates constraints whenever `isSubtype` or `leastUpperBound` are called. During inference, these methods are only called when they are used to enforce a check, usually for assignment or pseudo-assignment checks.

The previous inference system had two roots: one for the VarAnnot, and one for the target type-system annotations. This led to kludges to handle inconsistencies in the types. Dataflow had to handle types that only had one annotation specified. InferenceQualifierHierarchy had to handle both annotations being specified (which annotation should be used?). Having a single root avoids these issues.

(From Werner)
TODO: What should happen for over-constrained systems, e.g. if the programmer wrote annotations that cannot be fulfilled or if there is an actual bug in the source?

For inconsistent user-annotations:
instead of treating user-provided annotations as constants, they could be treated as special kind of equality constraints. A solver might break such an equality constraint if a better over-all solution can then be found.