Wesley Cox - coxw2
Grant Hughes - gyhughes
Joe Santino - jsantino

Compile-time warnings for Index Out Of Bounds Exceptions

Motivation

Runtime exceptions can cause a lot of problems in code. They can crash entire systems and lose important data. One of these exceptions is an index out of bounds exception. Our goal is to expand the type system to be able to guarantee that code will not throw that exception and crash. But instead it will be found at compile time and can be fixed. This would be extremely useful to developers because being able to prove that you code will not crash in this way before releasing it can avoid a lot of the problems associated with crashing programs. Such as annoyed users, losing money, and, in the case of medical equipment, even death.

Current manual solutions include using logic/weakest preconditions to prove the legality of index accesses and writing unit tests to check for illegal index accesses. Both of these are too time consuming and require careful attention of the developer. On the other extreme, a new type system could be developed by changing java's type system, resulting in a loss of compatibility with old code [2]. The main advantage to this approach is that it is pluggable; code does not need to be re-written to use this typechecker, only added to.

Approach

Our approach involves building a new type checker within the Checker Framework[1]. The Checker Framework is a tool used by developers to find and fix bugs in Java code. The checkers that use this tool finds the bugs at compile time, which saves a programmer's time by ensuring that bugs are found early on. We believe that we can create a plug-in to the Checker Framework that will aid programmers and help them identify and correct these bugs before they are compiled.

Our plan is to make the type checker as simple and as similar to possible to the default type checkers provided in the Checker Framework. Our current plan is to require developers to do the following to use our type checker:

1. Obtain the Checker Framework for the developer's environment.
2. Obtain the Index-out-of-bounds typechecker
3. Annotate desired indices with @IndexFor

Related Works

FindBugs is tool for java development that looks for bugs by doing static analysis on compiled Java byteCode. It does by default contain checks for “Array index is out of bounds” bugs. This program works by looking for bug patterns, or code idioms that are often errors [4]. Our project is different than FindBugs because instead of looking for patterns, it will use static analysis to generate warning. As we require the developer to write annotations into their code, we trade off developer time for a lower false negative rate.

Currently, It is very difficult to find a programming language that contains a typechecker that catches index-out-of-bounds errors at compile time. For pluggable frameworks such as Checker Framework or JavaCOP, It appears that no pluggable type system has been implemented to check for index-out-of-bounds errors. We do not have any evidence to prove or refute this, as the topic of checking for index-out-of-bounds errors is currently an unpopular topic. The general consensus of the community from many non-scientific programming websites (such as Stack Overflow) is that although trivial index-out-of-bounds errors are easy to catch, dynamic array size allocation and similar popular practices are too difficult to catch perfectly. To cope with this, we simply do not promise that false positives and true negatives will not exist. This is why we will only issue suppressible warnings unlike how Java’s type system denies compiling until the errors are fixed.

How it works

We will create a new type annotation called @IndexFor(String a). This will be used to annotate any int to be used as indices, and the value will be the name of the array it is for. This annotation will be the user facing part of this type system.

On the back end there will be a second type @indexHigh(String a) this annotation will say that this int was an index for an array but had 1 added to it and could be at the length of the array. This can not be used to access the array unless it is checked < a.length(or any value @indexFor(a) or @IndexHigh(a)) to turn it into @IndexFor again

There will a third type on the back end called @indexLow(String array) this will say that the int was an index for the array but was decremented by one and could be -1. This can not be used to access an array and must be checked to be >= 0(or any value @IndexFor the same array) to make it an @indexFor again.

Two more annotations will be @GTZero and @LTLength(a), if you check an unknown value is greater than 0 it becomes @GTZero, and if you check that it is less than @IndexHigh(a) it becomes @LTLength(a) any combination of these two checks on one value will result in it being @IndexFor(a).(due to being found in bounds)

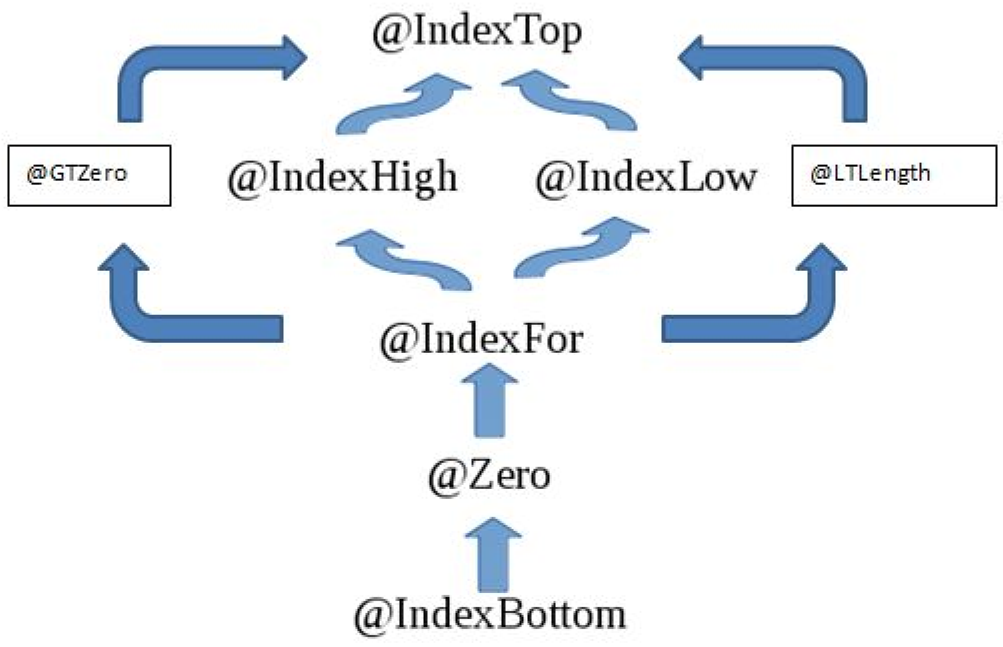
There will be a special annotation specifically for 0. @Zero, it will be a subtype of @IndexFor and valid for every array.

a.length is of type @IndexHigh(a)

If you check @Zero < @indexHigh(a) it becomes @indexFor(a)

Similarly with @Zero < @IndexFor(a) and @Zero > @IndexLow(a)

Therefore the type lattice will look like this:

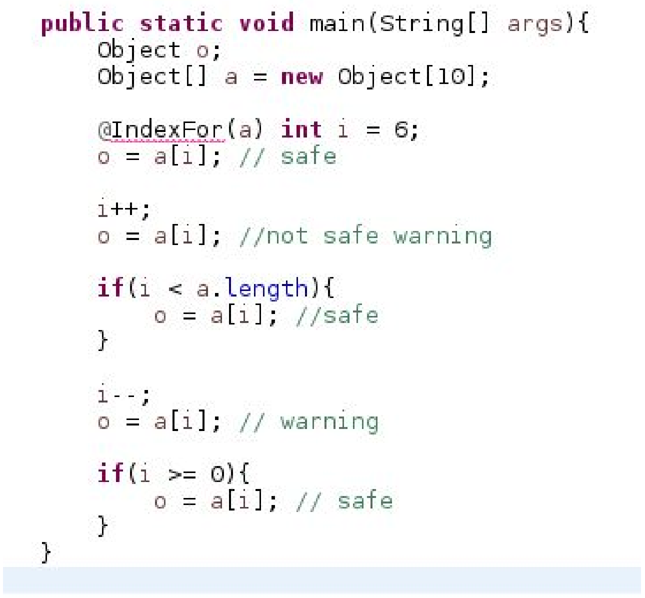


Ints will default to the @IndexTop annotation which is a supertype of all other annotations and cannot be used to access arrays.

Here are some arithmetic rules (transfer functions) for the types

- Multiplication on an @IndexFor(a) will make it @IndexTop
- Division by a positive number on @IndexFor(a) leaves the type unchanged
- Adding anything other than 1 to an @IndexFor(a) makes it @IndexTop
- Subtracting anything != 1 from an @IndexFor(a) makes it @IndexTop
- @IndexHigh(a) - 1 = @IndexFor(a)(to enable a.length -1)

Example:



```
public static void main(String[] args){
    Object o;
    Object[] a = new Object[10];

    @IndexFor(a) int i = 6;
    o = a[i]; // safe

    i++;
    o = a[i]; //not safe warning

    if(i < a.length){
        o = a[i]; //safe
    }

    i--;
    o = a[i]; // warning

    if(i >= 0){
        o = a[i]; // safe
    }
}
```

Challenges and Risks

The biggest risk to completing the project right now is learning to use the typechecker framework quickly. None of us have previous experience with type-checking code, so we need to learn what is and is not feasible to sanity check our design. If this can be done quickly enough, we then can spend adequate time actually building our type checker.

We are trying to mitigate this risk by first developing a really trivial type checker to learn the workings of the type checker framework. The current plan (subject to change) is to make a rudimentary @even checker to see marked integers remain even or not.

Additionally the checker has a risk of being too restrictive. If using this checker in your code causes developers to have to do to much work or to change their code to much they may not want to put in the work. Or if it restricts functionality of arrays it could cause problems with usability in some situations. In order to mitigate this risk we will use the case studies to see how well we can annotate without changing functionality. If necessary we could possibly weaken our guarantees (limit to certain situations) in order to warn for some errors while still be useable overall.

Plan

Given that this project is integrated into the CSE 403 class, the project must take 9 weeks and will only use resources open to public use or obtainable through the University of Washington. The typechecker framework falls under these resources.

Week	Plan
April 11 - April 17	Requirements specification // user side class hierarchy, use cases, user manual // find case studies // work on trivial checker
April 18 - April 24	Finish Trivial type checker (@even) (if not complete) // Software Design Specification // implementation plan(classes, interfaces)
April 25 - May 01	Skeleton implementation (Zero -feature) // Ability to add annotations and compile code with the annotation, although they don't do much yet
May 02 - May 08	Beta implementation // annotations work to point out possible out of bounds for arrays at compile time (still buggy) // work on list functionality
May 09 - May 15	Make sure beta passes all tests // Test more, fix issues, finalize beta release // List functionality // start case studies
May 16 - May 22	Create complete release // case study work
May 23 - May 29	finish case studies and evaluate effectiveness // code review
May 30 - June 05	Final release // demo

Evaluation

We are going to evaluate success of this project based on how difficult it is to add these annotations to code properly, how intrusive it is to the developer (ie. forced suppressions on false positives, requiring refactoring), and the amount of errors it can actually warn for.

Specific metrics will be:

- Number of warnings pre-annotation
- How many warnings were false(no possible exception)
- Number of annotations per hundred lines of code
- Number of methods/calls that needed to be refactored

- How long the annotations took us to insert (may not be as useful)

We plan to gather information using case studies based around these metrics. In these studies we would annotate source code for open source projects that we have access to and record all of this data as we do so. Possible case study projects would be:

In order to meet our goal, we should not be missing any possible out of bounds errors. Our checker should not allow index_out_of_bounds exceptions to be thrown assuming properly annotated source code. Defining success for how easy the annotations are to use is more difficult. As a baseline, a developer familiar with the java type system must be able to read our user manual and then properly use this checker on a medium-sized project correctly without much additional help.

By the middle of the course we expect to have the ability to annotate programs, have them still compile, and pass simple tests. And the expectation for the final product would be that we can show it helps prevent errors with proper warnings through case studies.

Case Studies

JDK's util package contains a Priority queue that uses an array to hold its ordering. It is just over 900 lines of code, about 50% being non-comment, whitespace, or closing brace lines. We expect a case study on this to only reveal our false positive count, given its wide and accepted usage.

Referenced works

[1] http://types.cs.washington.edu/checker-framework/
[2] Michael D. Ernst and Mahmood Ali, "Building and Using Pluggable Type Systems," in *ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Santa Fe, NM, 2010, pp.375-376
[3] http://groups.csail.mit.edu/pag/pubs/pluggable-checkers-papi-mengthesis.pdf
[4]University of Maryland, 'FindBugs™ Fact Sheet'. [Online]. Available: http://findbugs.sourceforge.net/factSheet.html. [Accessed:12-Apr-2016].