HW1
Talia Ringer
tringer

1. Proof Brittleness

Dependently typed languages like Coq are useful for writing rich specifications (*theorems*) about programs and proving programs correct with respect to the specification (writing *proofs* of those theorems). Researchers use these languages to verify everything from distributed systems[1] and compilers[2] to radiation therapy machines[3]. Unfortunately, verifying a large program right now often takes several years of collaboration between experts.

Changing a single theorem in these languages often breaks all other proofs that depend on that theorem – in other words, proofs are brittle. This is a consequence of the rich type system: Every theorem is a *full specification* of a type, and all proofs of the theorem are inhabitants of that type. This means that when you change a type of a dependency, it will change the type of other terms that use the dependency, and so the resulting proof will no longer prove the original theorem.

There is a lot of existing work on computer-aided verification, which often makes it easier to prove correctness of a program at the cost of opacity and predictability. These are useful alongside dependently typed languages, but are not a substitute. The core algorithms underlying many of these computer-aided verification tools (for example, Z3) are only useful for simple logics and do not extend to dependently typed languages[4]. Proof search techniques typically chop away at small sub-problems since the general problem is undecidable and a reasonable approximation of the general problem without admitting new assumptions is not known.

None of the existing work considers the *evolution* of theorems and proofs from one version to the next. In this problem, we know more than just the theorem we want to prove: We have an old theorem, an old proof, a new theorem, and a new proof. When a dependency on the theorem breaks, we can use the *differences* in the corresponding proofs and theorems to find a good patch that takes one theorem to the next. This is in many ways similar to work on patching in the software engineering world, except that we have a full specification (which in many ways is ideal, but which also makes the problem difficult since the type theory is so rich).

The approach I'm starting to take with Nate is to compile these proofs to a graph that highlights important semantic information (like the structure of inductive types) and use this semantic information to guide the search for a patch. I am also excited to see how existing patch work from the software engineering world relates to this problem.

2. Keeping Tests with Mocks in Sync with Code

Software engineers often use mock objects to stub out sensitive or irrelevant behavior and write safe and modular unit tests. Many frameworks exist to make writing mocks useful (for example, Mockito, EasyMock, and PowerMock), but these frameworks require programmers to specify behaviors in great detail and in the correct order. Naturally, this means that when the code under test changes (say, to switch the order of two service calls), unit tests often break even when the behavior of the code the under is identical.

This is a huge (and expensive) bottleneck for industry. During my time at Amazon, almost every unit

test I wrote made heavy use of mocks, and testing the code often took as long as writing the code to begin with.

The biggest challenge in automatically generating mocks and keeping them up to date is determining what the correct behavior should be. Existing work runs these tests without mocks, captures inputs and outputs, and uses them to introduce mocks automatically [5, 6, 7]. This is a great step forward, but interacting with the code under test even once to generate mocks can be expensive, can mask bugs in the code under test, and can compromise customer data. PBnJ [8] uses constraint solvers to simplify writing mocks, but requires programmers to write specifications.

One possibility is to treat the tests themselves as partial specifications rather than requiring the programmer to write specifications or examining the behavior of the mocks. That is, if the programmer writes tests that are correct initially, the tool can assume the behavior of these tests should still hold and use this assumption to generate mocks that preserve the behavior if possible. This will not always be correct (sometimes tests really should break), but when changes are small, it can be a useful assumption to guide a tool. Then the problem essentially becomes like other program synthesis and program patching problems, and techniques from both areas may be useful.

3. Migrating Code from Legacy Languages

The original Amazon code was written entirely in Perl, and developers spent years porting the system to different languages (mostly Java). Before this, the cost of porting some components was so high that developers opted to instead continue to work in Perl (which did not scale well) until upper-management made an executive decision to have developers port the code to Java. This was very slow and expensive. Many companies cannot afford to spend the money and time to do this to begin with, and so must work with code written in old languages that do not scale well or for which new expertise is scarce. It would be great to be able to just run a compiler that takes the code in the legacy language and spits out code in the desired language.

There are many barriers to making this tractable. The first is that often, these languages are very different (type systems, build systems, and so on), so writing a cross-compiler from one to the other that preserves behavior fully is impossible. Furthermore, some of these languages (like Perl) do not even have full specifications of their behavior. Finally, to be useful, the output code ought to be readable and maintainable by developers.

To solve this problem, we need to examine cross-compilation with underspecified source and target languages. Learning from examples (such as large code-bases before and after migration) may make this a more realistic problem to solve and may also help address readability challenges.

Sources

1. Verdi: https://homes.cs.washington.edu/~ztatlock/pubs/verdi-wilcox-pldi15.pdf
2. CompCert: http://compcert.inria.fr/
3. Neutrons: https://locore.cs.washington.edu/papers/ernst-neutrons.pdf
4. Congruence Closure in Lean: http://leanprover.github.io/papers/congr.pdf
5. Generating Test Cases for Programs that Are Coded against Interfaces and Annotations: http://dl.acm.org/citation.cfm?id=2544135
6. Automatic Test Factoring for Java: http://publications.csail.mit.edu/lcs/pubs/pdf/MIT-LCS-TR-991.pdf
7. Mock object creation for test factoring: http://dl.acm.org/citation.cfm?id=996838
8. PBnJ: http://web.cs.ucla.edu/~todd/research/pub.php?id=issta13