x86ISTMB Test Plan

We have developed an extensive test suite with many different kinds of tests to ensure the correctness of our system. This document describes our approach to testing and why we believe it demonstrates correctness.

1. Types of Tests

a. Snapshot

We tested many of our features using *snapshot testing.* Snapshot testing is a type of testing where the outputs of individual test cases are manually verified and then stored as files in the project. We have snapshot tests for the intermediate representation (IR) generator, the type checker/analyzer, and a few other pieces.

b. Property-based

We tested our core data structures, like directed graphs and what we call "context" (a fancy version of a stack), with QCheck. Data structures generally have a lot of properties that are easy to verify, making them a good choice for property-based testing. For example, in a directed graph, if a vertex *v* has an out-neighbor *u*, *v* should be an in-neighbor of *u*. Being confident in our data structures was important for us because debugging data structures while only seeing their effect on a compiled program would have been a nightmare.

c. Unit

Though we didn't rely extensively on unit tests, we did use them occasionally. Particularly, we used them to test complicated algorithms that are hard to verify with property-based testing, like register allocation or liveliness analysis. While we didn't impose strict coverage requirements, our tests were "glass-box" in the sense that we kept branches we covered and didn't cover in mind. The register allocation test suite, for example, was developed with Bisect coverage in mind.

d. End-to-End

We wrote many *end-to-end* tests to verify the correctness of the compiler as a whole. End-to-end tests run a program through the entire compiler, run the compiled output, and then check the output against the output of the *IR simulator* on the same program.

The IR simulator is a testing utility we wrote to run the intermediate representation of a program within OCaml directly, without compiling it. It is essentially an interpreter which we have high confidence in due to extensive manual and snapshot testing.

We run our end-to-end tests both with optimizations turned on and with optimizations turned off to verify that our compiler optimizations do not break program behavior.

e. Manually Tested Features

The command line interface (CLI) for our compiler was mostly tested manually. We didn't see a purpose in testing it automatically since we wanted to make frequent changes and since incorrect outputs are easily distinguishable from correct ones. Other than the CLI, all features of our compiler are tested automatically in some capacity.

2. CI Setup

Though it wasn't necessary for the project, our continuous integration (CI) setup was vital in giving us confidence in our compiler. On every commit to GitHub, the entire test suite is run on three versions of MacOS (with both x86 and Arm chips) as well as the latest version of Ubuntu. This helped us ensure that our compiler would work correctly on any platform.

3. Correctness Argument

We are confident in the correctness of our system due to our extensive test suite and CI setup. While we may not have perfect testing in all regards, we believe that we have tested our system to a sufficient degree.