6.945 Project Proposals

Jason Chen CJ Quines Matthew Ho

March 27, 2022

1 Property-based testing library

In software engineering, testing code is considered to be a vital part of delivering quality software. However, writing unit tests is extremely time consuming, and it is easy for the programmer to miss certain cases for more complicated functions. Therefore, it is desirable to try to automate this processes either partially, or entirely.

We build a QuickCheck-like tool that allows users to formulate properties of programs and then automatically runs tests based on the properties. Properties that users want to test will be specified via a DSL embedded in Scheme. The properties will be represented as functions along with typing information of the parameters, where the types will enable the generation of test data. To generate the actual test data, we choose random testing, with a uniform distribution over the type by default. This is interesting because it allows "generator deriving": we automatically have a way to generate lists of integers if we have a way to generate integers. We also allow users to specify their own distribution over types via generics to parallel real world data.

One potential extension to this idea is test shrinking: if a counterexample is found for a property, the library will automatically simplify the failure case to a minimal failing example via a built-in (and user-defined) rule system. This involves defining a generic shrinking function for each type, which is interesting in the case of recursive types.

2 Better Scheme errors

In a language intended for command-line, interactive use, error messages are an essential part of development. Scheme's error messages, however, leave much to be desired. The messages are often cryptic, saying only the error proper, without giving any context for where the error appeared or indication as to how it can be fixed.

We propose an extension for Scheme that provides better error messages. We aim to use Scheme's powerful introspection capabilities to include rich context for errors, including not just a stack trace, but row and column positions, what is expected and what is actually there, and hints about how to fix it, maybe even intended for beginners.

While Scheme's debugger is already a powerful tool, it can prove intimidating to use for beginners. Further, we believe it is possible to write errors that are effective enough to allow one-shot debugging: making a single change after seeing the error and fixing it without further interaction.

Modularity will be accomplished by factoring error message capabilities into several modules, which can be combined in various ways to produce errors of varying levels of detail. For example, one module could be responsible for the parsing and tagging of a source file, while another could be responsible for examining the stack. Thus, it would be possible to have interactive stack traces without having to place everything in a source file.

3 Board game move generation and search

In chess engines, there are two main components: search and evaluation. It is hard to generalize evaluation to other board games, as evaluation functions often rely on features specific to each game. However, search is much more generalizable.

As one increases search depth, the size of the search tree typically grows exponentially. This means that a board game engine must be able to generate legal moves extremely quickly in order to reach a reasonable depth, and must also have good ways to prune (for example, using alpha-beta pruning with some optimizations, and finding good ways to sort candidate moves so that they may be pruned more easily).

We propose to generalize the search component of a standard chess engine to other board games. Board games such as chess and checkers consist of a small number of types of pieces that have certain behaviors, such as captures and promotions. We will allow the user to specify the rules of a board game of their choice using a DSL, and create search functions that will take an evaluation function as input and return the list of moves and their corresponding evaluations.

This project will be divided into three main components:

1. Designing and implementing a DSL for specifying the rules of the board game
2. Writing code for fast move generation. This will involve lists of bitboards so that one can take advantage of the speed of bit operations.
3. Implementing algorithms to search the game tree, implement move ordering, etc.