Progress Report

Wesley Nuzzo

November 29, 2022

1 Repository

The repository for the project can be found at `https://github.com/UraniumCronorum/SC-FP`.

2 Original Compiler

The code for this section is in the folder 01_original_specs.

The new code is in hypothesis_R.py and hypothesis_tests.py. Both modules should be runnable by navigating to the folder and (with Python installed) typing "python [filename]", where [filename] is the name of the file.

2.1 hypothesis strategies

The result of running hypothesis_R.py should look something like this:

```
R.Program(R.Sum(R.Read(), R.Var('w')))
(program (+ (read) w))
R.Program(R.Int(0))
(program 0)
R.Program(R.Int(0))
(program 0)
R.Program(R.Var('r'))
(program r)
R.Program(R.Let((R.Var('a'), R.Let((R.Var('f'), R.Int(1197469428801169583)), R.I
(program (let ([a (let ([f 1197469428801169583]) -72)]) (- (let ([c (read)]) 10
```

This file contains code that generates a random program in source language for the compiler (which we're calling "R").

The implementation makes use of hypothesis's "recursive" strategy, which takes a base strategy and a strategy for extending that base strategy and generates recursive data.

As implemented, there's no sanity checking: resulting programs may (and usually will) have undefined variables, for example. The code generation for the extended compiler is more sophisticated.

2.2 unittests

The result of running hypothesis_tests.py should look like this:

```
....
----------------------------------------------------------------------
Ran 4 tests in 1.645s

OK
```

This file contains unittests that use the hypothesis generated code and test its properties.

The "RTest" tests contains a checkForm() test, which is more of a test that hypothesis' generated code is correctly formed, and a testEval() test, which is a true property test: it tests that running any program in the language results in outputting an integer.

"UniquifyTest" tests the first step in the compilation chain, "uniquify". As implemented, it simply checks that the resulting code is in the correct intermediate language (and is well-formed), and that running the code results in the same output as running the original code. The other requirement, that variables names do not get reused in new variable definitions, is currently taken care of by the implementation of R_uniq's "interpret" method.

A couple things to note: first, "R" allows user input, so this is handled by generating a random input stream and ensuring that the second execution receives the same data; second, compilation may fail on this step if a variable is used before it is defined. If the compiler rejects its input for this reason, the test ignores that case.

"PipelineTest" test each step in the pipeline, checking that the result of each step is in the correct language and evaluates to the same result as the previous step. If the compilation fails midway through, it checks correctness up until that failure. (As far as I'm aware, this should only happen if either a variable is not defined, or the program contains the input function, which has not been implemented in the later intermediate languages.)

3 Function Extension

The code for this section is in the folder 02_function_extension.

This code adds the ability to define and call functions in "R", and implements the first step in compilation for those function calls, including hypothesis-based unit tests for that step.

It also features more sophisticated code generation, including editing generated code to only used defined variables, and generating function calls with the correct number of arguments.

3.1 hypothesis strategies

The code hypothesis_R.py is quite a bit more complicated this time.

The function assignVarNames() corrects the problem of generated code using undefined variables. It attempts to rewrite those variables to use a defined variable and returns True if it succeeds and False if it fails (in case there are no defined variables yet).

The strategies provided are "simple_programs", which is basically the same as the implementation from before, "full_programs()", which adds function calls, and "safe_programs()" and "safer_programs()", which additionally try to make sure that all variables are defined before they're used.

3.2 unit tests

The file hypothesis_tests.py contains the unit tests for this section.

Some of these tests run rather slowly because safer_programs() takes a while to generate valid programs.

Most of the tests here are similar to the tests from the previous section, but differ in that they use the new strategies. Several tests are done with a variety of strategies.

Additionally, there's a test of the "repr()" method, which tests that the output can be read back in to be equivalent to the original object. This is a good illustration of the normal use of property-based testing, and could be a good basis for a test of a parser for "R" as well.

The current handling of infinite recursion is that we simply ignore the cases where evaluating exceeds Python's built-in recursion limit. I intend to do something more sophisticated later.