pyc-dbg

A Friendly Debugger for Our Compiler

Python -> Compiler -> x86

What did our compiler do?

A debugger that presents the source code and variables, but also the compiler intermediate language benefits us as compiler developers.

- We can see our compiler pass changes in "slow motion" after implementing them
- We can more easily isolate bugs to the compiler pass they originate from
- It feels good to watch your code run

Implementation

Stage 1:

Adding support for debugging to the compiler

Stage 2:

Implementing the debugging client

supporting debugging with our compiler

Three primary changes to the compiler:

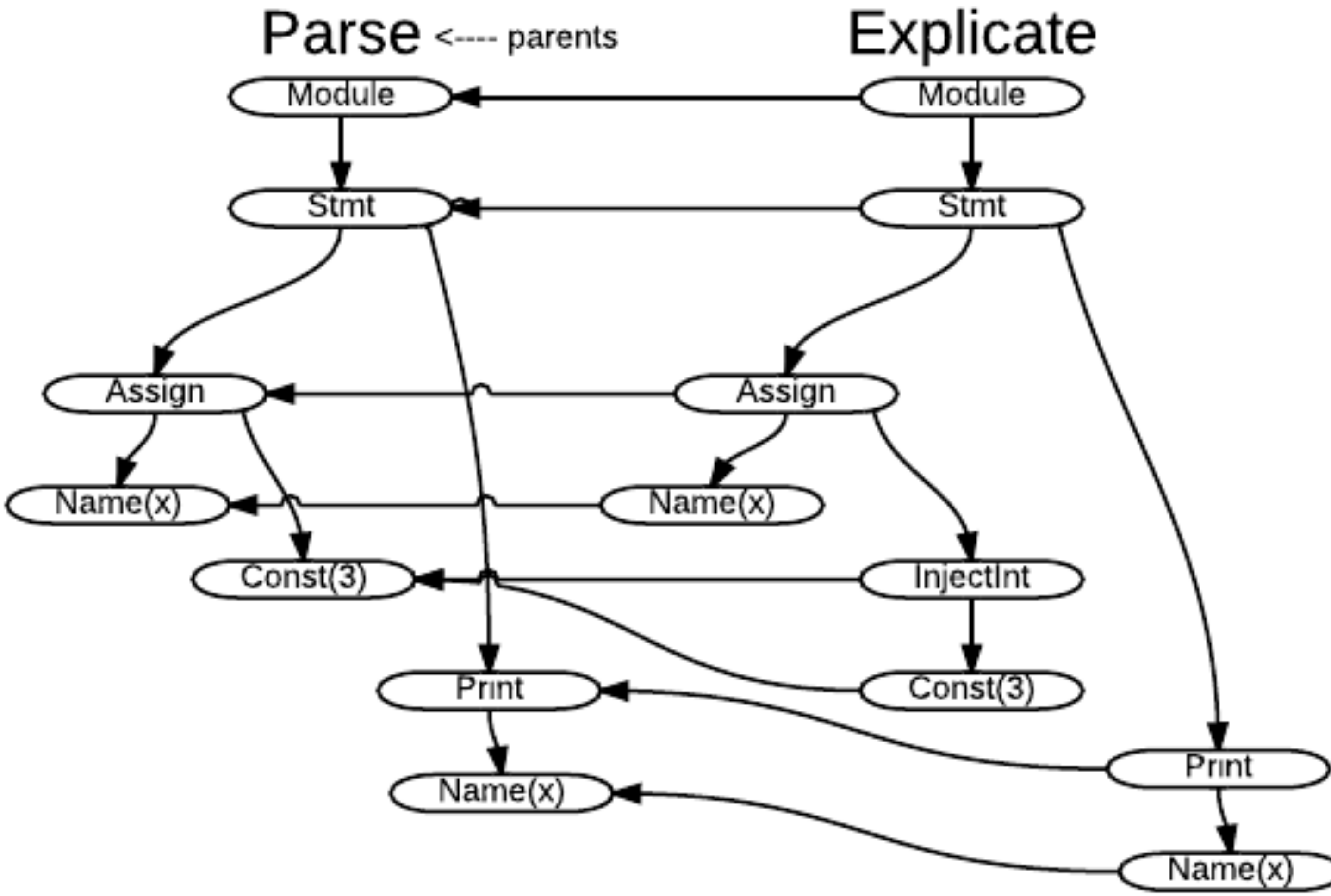
1. Track node lineage from each compiler pass so any AST node can be traced back to the output of ast.parse()
2. Support conversion of the "flatten" pass output to python code
3. Add a new section to the output assembly file to store debugging information

Node Lineage

Each compiler pass generates new AST nodes and drops the old ones, but we need to be able trace back to the genesis node from ast.parse() to know the source line number.

We keep track of this by storing the old AST node in a 'parent' attribute of the new AST node when we generate it.

An example of node lineage



IR to python conversion

Similar to other compiler passes: Use a visitor to recursively convert each AST node into its corresponding python text. Some example code:

```
def visit_BinOp(self, node):
    return "(%s %s %s)" % (
        pyc_vis.visit(self, node.left),
        pyc_vis.visit(self, node.op),
        pyc_vis.visit(self, node.right)
    )

def visit_Assign(self, node, **kwargs):
    print >>self.io, "%s%s = %s %s" % (
        self.tab_str(**kwargs),
        pyc_vis.visit(self, node.targets[0]),
        pyc_vis.visit(self, node.value),
        self.lineno(node)
    )
    return ""
```

.pyc_dbg ELF section

This section in the GNU assembly file contains everything the client needs to debug the assembled binary:

```
{
   ‘src’:           <string representing the original source code>,
   ‘sir_src’:  <string representing the generated SIR source>,
   ‘blocs’:    {
      <symbol name>:  {
         ‘src_lineno’:   <integer of the line that generated this function>,
         ‘sir_lineno’:   <same as above but referencing SIR source>,
         ‘mem_map’:      <dict of register allocation mappings>
         ‘insns’:   [
            {
               ‘src_lineno’: <integer of line that generated this instruction>,
               ‘sir_lineno’: <same as above but referencing SIR source>,
               ‘live’:       <the live set (before) of variables>
            },
            …  (more instructions follow)
         ]
      },
      … (more function blocks follow)
   },
   ‘name_map’: <dict mapping uniquified/heapified names to variables>
}
```

The above dict is serialized and hex encoded for storage.

The client

```
$> ./pyc-dbg -h
usage: pyc-dbg [-h] [-v] [-i INPUT] [-p CMD_PREFIX] file

debug binaries generated by pyc.

positional arguments:
  file                  file to debug

optional arguments:
  -h, --help            show this help message and exit
  -v, --verbose         print debug output.
  -i INPUT, --input INPUT
                        use input file when running binary
  -p CMD_PREFIX, --cmd-prefix CMD_PREFIX
                        prefix pyc related gdb commands with given argument.
                        default: "pyc"
```

How does it work?

The client exec's GDB and loads itself as a GDB python extension file.

On GDB startup, the python extension code parses the ELF binary using a library called 'pyelftools' and extracts the contents of the debug section.

It also uses the ELF symbol table to find the address and code size of each function block.

Finally, it waits for commands from the user.

Two interesting commands

`pyc-context:`

Displays current program context in terms of source code, IR source code, assembly code and live variables.

`pyc-step`:

Continues execution until the next logical source line.

Figuring out the context

Essentially, we derive the program context from just one thing: the program counter.

1. Check if the PC lies within the body of one of our function blocks.
2. If so, decode instructions from the function block start until PC. Let *n* be the number of instructions decoded.
3. Index into our instruction descriptors in our debug data structure:
   ```
   ins = dbg['blocs'][<blockname>]['insns'][n]
   ```
4. `ins` has the information we want: source line number, IR line number, and live before set.

Continuing to the next source line

A naive implementation:

1. Let the current line number be *i*
2. Read at least enough bytes at PC to decode the current instruction. Let this instruction be *x*
3. Set a breakpoint at PC + length(*x*)
4. Continue execution
5. When the breakpoint is triggered: If current line number *i2* != *i* then stop; else goto step 2

Are there any problems with this?

JMP exists...woops

If the current instruction is a JMP instruction our loop's breakpoint will never get triggered and the algorithm will fail.

Unfortunately, the GDB python API doesn't provide a callback event for when the value of the program counter is incremented, so we need to try a different strategy.

Brute force to the rescue!

This is the best I could come up with:

1. Let the current line number be *i*
2. Search our debug data structure for all instructions with line number not equal to *i*
3. Set a breakpoint at all of those instructions
4. Continue execution
5. When the breakpoint is triggered, clean up all the breakpoints that were set for this operation

Demonstration

When implementing p3, I made a mistake in the liveness algorithm for 'while' loops. This caused the regression tests grader_while1.py and grader_while2.py to fail.

Let's see if debugging grader_while1.py helps illuminate my error:

```
z = input()
x = 1
while z != 0:
    print x
    y = 2
    z = z + -1
print y
```

Questions? Comments?

If interested, you can find the code at:

https://github.com/cantora/pyc

THANK YOU ^_^