PyPE: a Python Pipeline Engine, Part 3

Part 3: The Back End

- Skeleton Updates
- A Flowgraph Intermediate Representation
  - Flowgraphs
  - Visualizing Flowgraphs
  - Topological Sort
- Optimization
  - Assignment Ellision
  - Dead Code Elimination

Skeleton Updates

As with the prior PyPE lab, you'll need new files. Some of these you'll copy over whole and some you'll need to extract functions from and copy into your files. (We realize this is a little tedious, but the alternative is forcing all of you to deal with `git` merge conflicts, which would be a bigger headache for you.) The easiest way to do this is to clone a new repo of the skeleton in a scratch location and then just copy over what you need. **Don't just blindly copy all the files.** You will overwrite your own work, so don't do that. We've put these files in the same skeleton repository as before: https://github.com/rdadolf/pype-skeleton (https://github.com/rdadolf/pype-skeleton)

Here's the list of files and functions you'll need for today's lab:

- `ast.py` : You'll need the new `mod_walk` function in `ASTNode` and the new `ASTModVisitor` class. These are slightly more powerful versions of the `walk` function you've already implemented.
- `error.py` : The whole file. This is a truncated version of the file I use internally.

Remember the import error in the first assignment? This was the culprit. Feel free to expand upon it.

- `fgir.py` : The whole file.
- `optimize.py` : The whole file.
- `parser.py` : We've given you implementations of the operator expressions (e.g., `p_op_add_expression` , etc.). You probably already have a working version, but you should make sure that it makes to the same ASTID as these versions. You may need to modify the `ASTEvalExpr` constructor call if you've written it differently. The salient point here is that the `ASTEvalExpr` 's `op` property must map to an `ASTID` which matches the python built-in method of its operator. You'll also want to tag these methods in your timeseries class with the `@component` decorator you wrote. (If you don't do this, your compiler will complain about undefined symbols whenever you use an operator in a .ppl file from now on.)
- `pipeline.py` : You have a choice on this. If you haven't modified the original file (or you don't mind losing your changes), then **the easy way is just to overwrite the old file with the new one**. If you added code, you'll need to merge the two files manually. To help you out, you can check the file history on github (https://github.com/rdadolf/pype-skeleton/commits/master/pype/pipeline.py). This gives you a list of all the changes.
- `semantic_analysis.py` : Copy the `CheckSingleIOExpression` and `CheckUndefinedVariables` functions. There are new semantic checking functions we've written for you. It should help to catch some errors a little earlier in the process and make their error messages clearer.
- `symtab.py` : You'll need the new `lookupsym` function. This is a scoped version of name resolution, similar to the one you saw in the stupidlang environments earlier in the class.
- `translate.py` : You'll need the new `LoweringVisitor` function as well as the new `import` statement at the top (the ones for `fgir` and `error`). This is a pretty lengthy visitor class that does the heavy lifting of turning an AST into FGIR. You'll be pleased to know that I was originally planning on having you write this function.

This is a lot of changes, so please read back over them carefully. If you start running into bugs in a function you didn't write or new bugs in a function that previously worked, make sure you didn't accidentally skip one of these steps.

Final Note:

Back to top ↑

A Flowgraph Intermediate Representation

The back end of a compiler truly starts when an AST is converted to some intermediate representation or "IR". The purpose of an IR is to reduce the functionality of a user's program down to its most basic parts. For PyPE, this means a computational graph which describes how data flows between operators declared in your program.

PyPE's IR is called FGIR (pronounced "figure"), and it consists of two major pieces: a top-level structure which keeps track of components, and graphs for each component called "flowgraphs". The top level is really little more than a glorified dictionary, but it also provides us a convenient place for interfacing with optimization passes, which we'll describe later. The meat of a FGIR object is in its flowgraphs.

Back to top ↑

Flowgraphs

A flowgraph is simply a graph which represents the data dependences in a PyPE program. For instance, if you write these statements:

```
{ example
  (input x y)
  (:= z (+ x y))
  (output z)
}
```

then we say that `z` has data dependences on `x` and `y`. This should be intuitive: you can't compute `z` without knowing the values of `x` and `y`. A flowgraph captures these relationships explicitly.

Flowgraphs are *DAGs*. If you recall from the lecture on graphs, directed acyclic graphs just means that (1) edges have direction, and (2) there are no cycles (so if you start at any given node, there's no path you can trace by following edges which leads back to itself). Additionally, we implement flowgraphs with adjacency lists, with the minor tweak that edge lists are held by the graph, not by individual nodes. The best way to understand is simply to look at the implementation provided in `fgir.py`.

Nodes in a flowgraph are given unique symbolic names. This might seem odd at first: in the earlier example, why can't we just name the nodes by their variable names (`x` , `y` , `z`)? The answer is that there are actually more graph nodes! It's easier to see when considering a slightly larger example:

```
{ example2
  (input x y)
  (:= z (+ (* x x) (* y y)))
  (output z)
}
```

The `+` expression depends on the result of the `(* x x)` and `(* y y)` expressions, and since flowgraphs are supposed to capture data dependencies between expressions, how do we represent this? The answer is, as we hinted at, by giving these "anonymous" expressions unique names. We take this one step further and give *every* graph node a new, unique name and then tag these nodes with a label if they have a variable name in the original program. This renaming is actually fairly common in compilers, graphs, and programs in general, as it abstracts your functionality (a dependence graph, in this case) away from the source representation (a .ppl file).

Back to top ↑

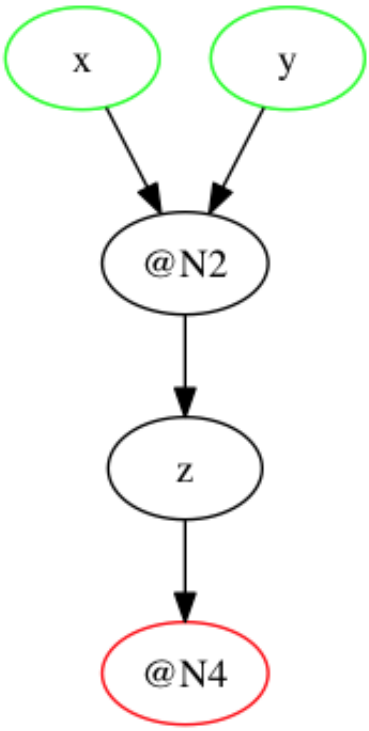
Visualizing Flowgraphs

This is a lot to take in, and so far we haven't had you do anything, so let's change both.

Flowgraphs come with a function built-in called `dotfile` . This function outputs a string representation of the flowgraph in "dot" format, suitable for a program called graphviz. Graphviz (http://www.graphviz.org/) is a program originally developed at AT&T in the late 90's to layout graphs. Since then, it has become the de facto standard for simple graph visualization, and its "dot" format (for directed graphs) is an extremely popular, portable representation for tools which automatically need to emit graphs. The power is largely in its simplicity: dot is a text-based format which is human-readable and fairly easy to learn. Consider the dot representation of our first example:

```
digraph example {
  "@N3" -> "@N4"
  "@N0" -> "@N2"
  "@N1" -> "@N2"
  "@N2" -> "@N3"
  "@N0" [ label = "x" ]
  "@N1" [ label = "y" ]
  "@N3" [ label = "z" ]
  "@N0" [ color = "green" ]
  "@N1" [ color = "green" ]
  "@N4" [ color = "red" ]
}
```

Using the `dot` program, you can render this as:



The syntax for `dot` can be found here: dot documentation (http://www.graphviz.org/pdf/dotguide.pdf), but the syntax for the command that generated the above picture is just this (for a png file):

```
dot input_file.dot -ooutput_file.png -Tpng
```

We'd like you to get your flow up and working to the point where you can dump out dot descriptions of the sample programs in pype-skeleton. (You may need to comment out unimplemented code to do this.) Then, download and install graphviz and render a graph of

the FGIR.

You do not need to commit the images to your repo, but this will be a useful debug tool later.

Note: if you cannot get graphviz installed, you can use a web version here: WebGraphViz (http://www.webgraphviz.com/). Just copy-paste.

Back to top ↑

Topological Sort

You've already learned about topological sorting in the graph lecture. If you need a refresher, feel free to look back at the notes or google some references.

You'll also notice that there's a stub for a topological sort in the `Flowgraph` class. Please implement that now. The output should be a topologically-sorted list of node ID's (not the nodes themselves). You should feel free to write helper functions if you need them.

Back to top ↑

Optimization

Optimization is one of the huge selling points of a compiler. When you write:

```
int f(int a, int b) {
  if(a%2==0)
    return (a+b)/2;
  else
    return (a+b-1)/2;
}
int main() {
  return f(20,10);
}
```

The compiler won't even bother to generate code for any of that (in its most aggressive mode). It will generate a program which has one instruction: return 15. This is because by analyzing the program carefully, it can prove that the two are equivalent (and it has been programmed to favor the faster version).

Now that your PyPE compiler has a working front end and can generate an intermediate representation, it's time that we teach it how to optimize your PyPE programs as well.

Back to top ↑

Assignment Ellision

One of the first things that people unfamiliar with compilers learn is that many optimizations aren't really all that clever. Instead, they do stupid things like "if there's a test if a number is zero and it's testing a non-zero number, just remove the test." Many people think, "why would you even write code like that to begin with?" The answer is that most people don't... directly. Instead, code like this comes from a long sequence of optimizations strung together. At some point, a sequence of optimization might prove that the variable the programmer was actually testing is always zero. At that point, it substitutes a zero into the conditional, and the program is left with a silly-looking test. But it's exactly this sort of repetitive, mechanical reduction that compilers excel at, and it's the reason why there can be orders of magnitude difference in performance between unoptimized code.

Optimization also is a good way of cleaning up after other parts of the compiler. In the case of PyPE, the `LoweringVisitor` transformation which actually produces the FGIR from an AST in the first place is rather dumb: whenever it sees an assignment statement, it inserts a new graph node. This is simple (and a good way of avoiding bugs), but it leaves us with extraneous graph nodes that really do nothing. If you look back at the visualization from our example, you'll see exactly this: the result of the sum is the node `@N2`, but there's another node `z` immediately after it which simply passes the value forward to the output node `@N4`. We want you to eliminate `z`.

Specifically, we want you to write a subclass of the `FlowgraphOptimization` class which finds all the assignment nodes (conveniently, they all have type `FGNodeType.assignment`) and remove them. In order to preserve the program semantics, you'll need to take the nodes which depend on those assignment nodes (`@N4`, for instance) and modify them so that they point to the node that the assignment node depended on (`@N2` in this case). Because assignment statements only take one expression (you can check the parser grammar to prove this to yourself), you're guaranteed that there's always only one node that the assignment node depends on. Finally, we want you to clean up the variable name mappings. Recall that every node is assigned a unique name, and in this case, the node which shows up as `z` is actually called `@N3`; we just know it as `z` because of the variable mapping table

that the flowgraph holds. So while we're removing the assignment node `@N3`, we'd also like to re-map the variable `z` to the predecessor of the assignment node we just removed: namely, `@N2`. So you'll need to update the flowgraph's `variable` attribute to have the string "z" map to the node ID `@N2`.

Back to top ↑

Dead Code Elimination

Finally, we want you write another subclass of `FlowgraphOptimization` which removes dead code.

"Dead code" is code that doesn't affect output. For instance:

```
{ component
  (input x)
  (:= useless 1)
  (output x)
}
```

The assignment statement does nothing here. In order to make our program faster, we'd like to remove it entirely, so it doesn't consume resources.

Specifically, we'd like you to eliminate all *non-input* graph nodes which are not reachable (https://en.wikipedia.org/wiki/Reachability) from any output node by following dependences.

We've given you a rather complicated example of dead code in the `six.ppl` sample which you can use to test. Don't forget to remove references to variables whose nodes no longer exist.

Back to top ↑