Assignment 4: type analysis

ECEN 4553 & 5013, CSCI 4555 & 5525
Prof. Jeremy G. Siek

The main concepts of this assignment are

- using recursive functions to analyze the program,
- understanding the overhead associated with dynamic typing and run-time polymorphism.
- optimizing by type specialization

1 Motivation

Look at the C code produced by your $P_1$ compiler for the following program.

```c
print 1 + 2
```

Compare it to the C code produced by the version of your compiler for $P_0$. It doesn’t look very efficient does it? By introducing general mechanisms for dealing with polymorphism, we have introduced a lot of overhead: creating tagged objects, checking tags, and dispatching to different functions. But often times a program, such as the one above, does not need polymorphism. For this assignment we make our compiler smarter so that it avoids the overhead for polymorphism when it is not needed.

To produce better code, we need predict what type of object a given expression will produce. For example, in an addition expression such as $e + e'$, where $e$ and $e'$ are arbitrary expressions, if we can predict that both $e$ and $e'$ always produce integers, then we can implement the addition with just $+$ in C. The process of predicting the type of object produced by an expression is called static type analysis.

Unfortunately, in dynamic languages like Python, an expression may sometimes result in object of many different types. Returning to an example from the previous assignment, the following expression sometimes returns an integer and sometimes a float.
3 if randint(0,1) else 3.14159

In these cases, the type analysis must make a conservative decision. One way to do this is to say that the type of the expression is the union of all Python types, call it `pyobj`.

2 Type Analysis

We could implement the type analysis by extending one of the existing passes in the compiler to keep track of types, but that would complicate that function. A good software engineering principle is to have every function perform one specific action, not two or more. Thus, a better approach is to write a separate type analysis function that annotates the AST nodes with the predicted type. The types are: `int`, `float`, `bool`, and `pyobj`. Figure 1 provides an incomplete sketch of the type analysis function. The function recursively traverses an IR AST, predicting the type of the object for each node. The parameter `env` (for environment) is a dictionary that maps variables (from `let` nodes) to their type. The auxiliary nested dictionary `op_returns` is used to determine the appropriate return type for the primitive operation. The assignments to the `.type` attribute take advantage of the ability of Python to add attributes to objects on the fly.

3 Type Specialization

Now that we have produced predictions about the type of each AST node, we can use that information to compile primitive operations into specialized versions of the operations for particular types. For each primitive operation, such as `add`, we’ll create specialized versions for when the specific type of arguments are known. (We’ll just use the unspecialized version when one argument is known and the other is unknown.) Name the specialized primitives based on the type they are specialized for, so `add_int` is the specialized version of `add` that takes two integers and returns an integer.

An important invariant that we need to maintain is that the compiler produces AST nodes with the same type as what was predicted by the type analysis. For example, in an `IfExp`, when the types of the two branches do not match, we’ll need to produce code that converts the objects to `pyobj`. For this purpose you can use `PrimitiveOp` AST nodes, using `make_int`, `make_float`, or `make_bool` for the function name as appropriate.
Figure 1: Incomplete sketch of the type analysis function.
Note also that in and IfExp, if the type for the test expression is known, then we can leave it as is. If type for the test expression is unknown, i.e., pyobj, then we need to insert a primitive operation that will test whether the pyobj is true. In the previous assignment we always had to insert this test.

**Exercise 3.1.** Implement type analysis and type specialization for $P_1$. The type specialization pass should take as input an $IR_0$ AST and return a new $IR_0$ AST. Check that your compiler still passes all of your tests for the $P_1$ language.

![Diagram](image)

Figure 2: Overview of the compiler organization.