

# Assignment 4: type analysis

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

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

```
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 *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`.

## 1 Type Analysis

We could implement the type analysis by extending our recursive compilation function to keep track of types, but that would complicate that routine. 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 the AST, predicting the type of the object for each node. The auxiliary dictionary `return_type_plus` is used to determine the appropriate return type for addition. The assignments to the `.type` attribute takes advantage of the ability of Python to add attributes to objects on the fly.

The call to `predict_type` should be placed in your compiler before the call to the function that produces the C code.

```

def predict_type(n):
    if isinstance(n, Const):
        if isinstance(n.value, int):
            n.type = 'int'
        elif isinstance(n.value, long):
            n.type = 'int'
        elif isinstance(n.value, float):
            n.type = 'float'
        else:
            print 'unrecognized kind of constant'
    elif isinstance(n, Name):
        if n.name == 'True' or n.name == 'False':
            n.type = 'bool'
        else:
            print 'unrecognized name'
    elif isinstance(n, Add):
        predict_type(n.left)
        predict_type(n.right)
        n.type = return_type_plus[(n.left.type, n.right.type)]
    elif isinstance(n, IfExp):
        predict_type(n.test)
        predict_type(n.else_)
        predict_type(n.then)
        if n.else_.type == n.then.type:
            n.type = n.else_.type
        else:
            n.type = 'pyobj'
    elif ...

```

Figure 1: Incomplete sketch of the type analysis function.

## 2 The Type Specialization Optimization

Now that we have produced predictions about the type of each AST node, we can update the code generation to use this information. First, we no longer need to convert constants to pyobj right away. Instead, they will be converted as necessary in other places. For example, the case for Const can simply return a string that is the C code for the constant.

```
def p1_to_c(n):
    if isinstance(n, Const):
        return repr(n.value)
    ...
```

An important invariant that we need to maintain is that the compiler returns a piece of C code that produces that same kind of object 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. Assuming that our runtime support code includes functions called `make_int`, `make_float`, etc. that create pyobj's out of integers, floats, etc., we can do the following:

```
def convert(e, t):
    return 'make_' + t + '(' + e + ')'
...
def p1_to_c(n):
    ...
    elif isinstance(n, IfExp):
        c_test = p1_to_c(n.test)
        c_else = p1_to_c(n.else_)
        c_then = p1_to_c(n.then)
        if n.test.type == 'pyobj':
            c_test = 'is_false(' + c_test + ')'
        if n.else_.type != n.then.type:
            c_else = convert(c_else, n.else_.type)
            c_then = convert(c_then, n.then.type)
        return '(' + c_test + ' ? ' + c_else + ':' + c_then + ')'
    ...
```

**Exercise 2.1.** Implement type analysis and the type specialization optimization for  $P_1$ . Check that your compiler still passes all of your tests for the  $P_1$  language.