TYPE INFERENCE AND LACKWIT

GILBERT BERNSTEIN

1. Type Checking

I'm going to use a running example to make this easier to follow

```
f : (int, int, int) -> int
let f x y z =
    if x > 0 then
        x = y
    else
        x = z
```

Suppose we're compiling this code. The first thing we'd do is parse it into an abstract syntax tree. Here is the AST for the function `f`:

AST PIC

Notice at the beginning of the code snippet provided, there is a type signature:

```
f : (int, int, int) -> int
```

The goal of type *checking* is to ensure that the programmer is not lying when they make statements like `f` is a function that takes three `int`s and returns an `int`. Now suppose that someone gave us (totally out of the blue) a complete labeling of our AST with a type for every node. We might write down these labels like this

TYPED AST PIC

where `x : int` is a label on the expression `x` meaning that the expression `x` has type `int`. Now our task of type checking is very simple. All we need to do is make sure that all of these labels are consistent with each other, according to some set of *typing rules*. These typing rules are usually written like this:

$$\frac{e_1\colon \texttt{bool} \qquad e_2\colon \tau_1 \qquad e_3\colon \tau_1}{\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3\colon \tau_1}$$

In english, we would read this rule as follows. Given that we know the following: that e_1 is an expression of type `bool`, that e_2 is an expression of type τ_1, and that e_3 is an expression of type τ_1, then we can conclude that the compound expression `if` e_1 `then` e_2 `else` e_3 has type τ_1. More importantly, notice that if we just flip this rule upside down, superimpose it on our AST and substitute in the specific value `int` for the placeholder τ_1 then we get a perfect match with our labeling.

In reality, we are not handed a complete labeling of our AST, so we have to generate it ourselves. We would start with our goal. To prove that the if-then-else statement has type `int`:

PARTIAL AST PIC

Then, we would reason in reverse that the sub-expressions must have the following types:

PROPAGATED LABELING AST PIC

Obviously, doing this gives us a simple recursive algorithm to type-check with. However, look what happens when we reach the leaf node `x`:

BOOLEAN EXPRESSION X BRANCH ZOOM

We need some way to be sure that `x` is of type `int` here, but in general `x` might be any type. To solve this problem we need to make some *assumptions* (to use Milnor's language). In modern parlance, we want a *type environment* that records information like "`x` has type `int`", and makes it available to us when needed.

Notice that when we gave the type checker a signature for function `f` to check, we told it that `f` takes three `int`s. Using this information, we know that `x`, `y`, and `z` have type `int`. What we really want the type checker to do is not to prove that the if-then-else statement has type `int`, but to prove that the if-then-else statement has type `int`, *assuming* that `x`, `y`, and `z` have type `int`. We write this statement in symbols as

$$\mathtt{x} : \mathtt{int}, \mathtt{y} : \mathtt{int}, \mathtt{z} : \mathtt{int} \vdash \mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3 : \mathtt{int}$$

We use these assumptions as input to the type checking algorithm. This makes a good mnemonic. The input goes to the left of the expression, and the output to the right. By passing these assumptions down the tree, and adding new assumptions whenever we declare a new variable, we can ensure that we always have the correct type on hand to assign to any given variable name.

IN MULTIPLE STEPS: BOOLEAN EXPRESSION X BRANCH RESOLVED

Now, when we reach the leaf node for `x`, we've carried along our assumption that `x` has type `int`, so we can easily assign `x` type `int`

2. Type Inference

In the last section, we had to annotate our function with a type signature by hand in order to type check it. What if we didn't do anything of the sort and just wrote out the function:

```
let f x y z =
    if x > 0 then
        x = y
    else
        x = z
```

The goal of type *inference* is to figure out what the type of a function like `f` is, even though the programmer didn't tell us explicitly. Based on the program text the programmer did write, we can probably guess (or *infer*) what type the programmer meant for `f` to have.

Just like with type checking, we start with an AST

VANILLA EXPRESSION ONLY AST PIC

But this time, when we go to start labeling the AST, we have no idea what types to assign to anything. So, we introduce *type variables* like $\alpha, \beta, \gamma, \delta$ to use as placeholders. For instance, our previous type environment, aka. our set of assumptions is now provided as

$$\mathtt{x} : \alpha, \mathtt{y} : \beta, \mathtt{z} : \gamma \vdash$$

However, note what happens when we get down to the `x > 0` branch of the AST.

PICTURE OF BRANCH

We can only compare two variables if they have the same type, so since `0` has type `int`, `x` must have type `int` too. But, we've already assumed that `x` has type α. What can we do? Somehow we have to account for these unanticipated *constraints*.

Here's an idea. How about we just go ahead with the algorithm and give `x` type `int`. Then, rather than deal with the constraint that $\alpha = \mathtt{int}$ right now, we'll just write it down and *deal with it later*. This is the key idea for Milnor's algorithm W. All we have to do is extend our previous type checking algorithm by returning a set of accumulated constraints.

To make this idea more concrete, consider our previous version of the if-then-else expression typing rule:

$$\frac{A \vdash e_1 : \mathtt{bool} \qquad A \vdash e_2 : \tau_1 \qquad A \vdash e_3 : \tau_1}{A \vdash \mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3 : \tau_1}$$

In order to convert this rule to support constraints, we want to replace all of our types with distinct type variables, and move any particulars into constraints:

$$\frac{A \vdash e_1 \colon \tau_1 \qquad A \vdash e_2 \colon \tau_2 \qquad A \vdash e_3 \colon \tau_3}{A \vdash \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \colon \tau_4}$$

with the constraints that $\tau_1 = \texttt{bool}$ and $\tau_2 = \tau_3 = \tau_4$. If we further account for constraints generated by inference on the sub-expressions e_1, e_2 and e_3, then we get the following, complete, type inference rule:

$$\frac{A \vdash e_1 \colon \tau_1; C_1 \qquad A \vdash e_2 \colon \tau_2; C_2 \qquad A \vdash e_3 \colon \tau_3; C_3}{A \vdash \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \colon \tau_4; \tau_1 = \texttt{bool}, \tau_2 = \tau_3 = \tau_4, C_1, C_2, C_3}$$

which is quite a mouthful if you try to say it in English!

The general form of the annotated AST nodes we're working with has now expanded to encompass four different terms.

$$A \vdash e : \tau; C$$

These are, the type environment A (in modern parlance, or assumptions in Milnor's language), the expression e, the type τ, and constraints C (which Milnor models as a substitution mapping).

Using this labeling notation with selective omissions, we can visualize the result of running type inference on the AST for `f`:

SHOW CONSTRAINTS GENERATED BY TYPE INFERENCE

A second algorithm called *unification* is then run to find the most general possible substitution of variables that satisfies the produced set of constraints. In this case we get a result like the following out of unification.

RESULT OF UNIFICATION

Finally, we take the result of unification and use it to populate our AST. This tells us what the types of everything ought to be.

AST POPULATED BY UNIFICATION

To keep things familiar, I constructed this example to produce the same type signature that we got in type checking. However, this is not generally true. Type inference often produces more general types than we may have intended. For instance, suppose that I changed the conditional in `f` to check equality between `x` and `z`:

```
let f x y z =
    if x == z then
        x = y
    else
        x = z
```

Now f doesn't contain anything that specifically refers to integers. Consequently, the inferred type signature won't refer to integers either. However, because all of our variables are potentially compared or assigned to each other, we may infer that they must all be of the same type, generating the signature

$$\texttt{f : } (\alpha, \alpha, \alpha) \texttt{ -> } \alpha$$

where α is a stand in for any type.

This is the idea behind generic types, and type polymorphism.

3. Lackwit

Lackwit takes the idea of a type inference algorithm and repurposes it to try and infer other properties of programs not traditionally represented as types. For instance, in our running example type inference was able to determine that the input variables x, y, and z must all share the same type on the basis of interactions between those variables. Suppose that x, y, and z were actually all pointers to integers. Then we could use the result of type inference to conclude that x, y, and z might be aliased, on the basis that they are all given the same type. Lackwit is all about using type inference to draw conclusions like this.

To clarify this point, consider this modification to our function f:

```
let f x y z w =
    if x == y then
        z = w
    else
        w = z
```

Type inference would now produce the following signature:

$$\texttt{f : } (\alpha, \alpha, \beta, \beta) \texttt{ -> } \beta$$

From this we can conclude that f does not mix the first two arguments with the second two. Even if all four arguments happen to be of the same type, we can benefit from pretending that they're not. Lackwit's original goal and proof of concept was to run on C code, so let's consider a C version of the function f:

```
int* f(int* x, int* y, int* z, int* w) {
    if(x == y) {
        z = w;
        return z;
    }
    else {
        w = z;
        return w;
    }
}
```

Lackwit's first order of business is to completely ignore the stated types of any variable. Then, type inference is performed. Finally, Lackwit reinterprets these results to achieve a partition of the original types into more specific types, which are referred to as different *representations*. In the above function, we end up producing the annotation:

$$\texttt{int*}_\beta\ \texttt{f(int*}_\alpha\ \texttt{x, int*}_\alpha\ \texttt{y, int*}_\beta\ \texttt{z, int*}_\beta\ \texttt{w)}$$

This signature provides a concise, parameterized summary of what `f` might do with its arguments. By modifying the primitive typing rules provided for basic statements like assignment, we can use type inference to infer different properties of a program.