CIS 425 - Week 5 - Lecture 8

Alyssa Kelley
Anne Glickenhaus
Miguel Nungaray

29 October 2019

1 Type Checking

Definition 1. *Type checking is the process of verifying and enforcing the constraints of types, and it can occur either at compile time (statically typed languages) or at runtime (dynamically typed languages).*

Type checking is all about ensuring that the program is type-safe, meaning that the possibility of type errors is kept to a minimum.

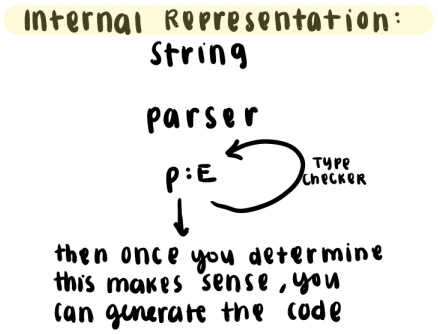
Example 2.

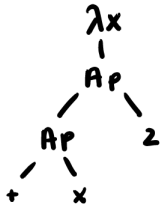
$$fun\ f(x : int) = x + 2;$$
$$type : int \to int$$

So given the $fun\ f$, the type checker will be able to say that the function has $type : int \to int$.

1.1 Implementing a type checked

When you type in your program at the REPL, your program is read as a String. The parser turns that string into an internal representation of your program, which is described by a datatype, say E. Given this internal representation, you can now write the type checker. The type checker analyses E, if it determines that your program is type safe then you can do code generation, and you are guaranteed that your program will not core dump or produce a segmentation violation. Before doing code generation, you could also optimize your program in terms of rewriting programs of type E into a new program of type E. You can see this process in figure 1.





Example 3. Let us now declare our function f as follows:

$$fun\ f(x : real) = x + 2.0;$$
$$type : real \to real$$

Notice how the type has changed, even though we use the same symbol +. We call + an *overloaded* symbol, since the same symbol stands for different operations. Instead, consider

$$fun\ f(x) = x + x;$$

which type should one generate? Since the user did not give enough information, one possibility is to raise an error. ML instead opt for a different solution: it assumes that x is of type int.

2 Type Inference

Example 4.

$$fun\ I\ x = x;$$

In this function definition we don't know what they type of x is. What type would be generated in this case?

Definition 5. *Type Inference refers to the automatic deduction of the type of an expression in a programming language.*

So in example 4, you have to infer the type of the input and output, which in this case must be the same and could be any type. The goal of a type inference system is to find the *most general type MGT*. In this case the MGT is $\alpha \to \alpha$. So our goal is to explain how a type inference system works.

2.1 How to write a type inference system

- Have a representation of your program. (Abstract Syntax Trees)
- Assume that overloading is not allowed.
- We will write our program in curried form.
- We will use prefix notation.
- Assume that the type + is curried.

Example 6.

$$+ : int * int \to int \tag{1}$$
$$+' : int \to int \to int \tag{2}$$

Here we define two forms for the type of +. Form (2) is the curried version of form (1). This means it takes the arguments one at a time. It allows for partial application.

Example 7.

$+\,(2\ 3);$	$(Correct)$
$+\ \ 2;$	$(Wrong)$
$+'\,(2\ 3);$	$(Wrong)$
$+'\ \ 2;$	$(Correct)$

We can see that depending on the form you can only pass in a certain number of parameters.

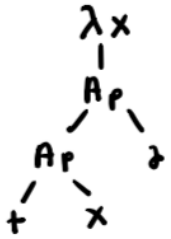


Figure 1: Representation of the program in example 8

Example 8. Consider the following definition:

$$fun\ f\ x = \ +\ x\ 2;$$

Suppose you don't want to have variables in your representation, because we know that variables are confusing, internally. How could we eliminate the variables and still maintain the correspondence between the variable and the lambda node? One idea is to just use a back pointer, as in figure 8.

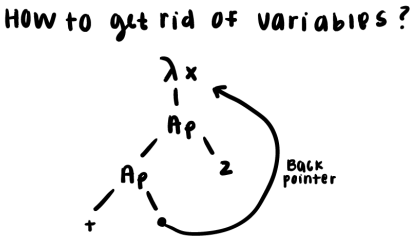
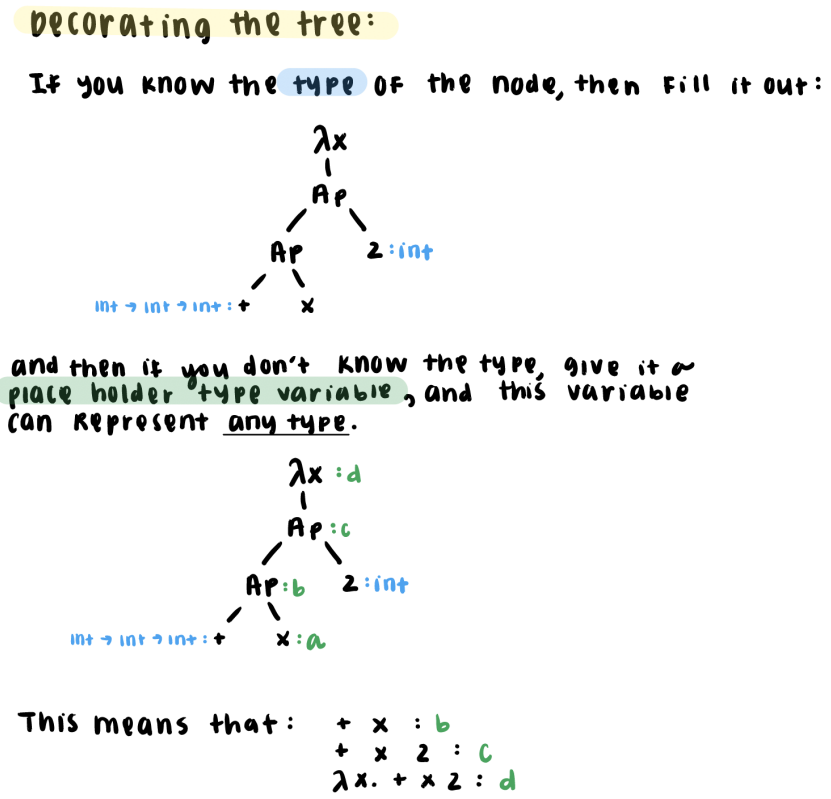


Figure 2: Using backpointers instead of variables

2.2 Decorating the Tree - Continuing from AST

So now that we have an abstract syntax tree, what do we do?

- The first phase is to navigate the tree and for each node:
  - If you know the type of that node, you decorate that node with the type.
  - If you do not know the type of the node, then you give it a placeholder variable.



This tree is now showing the type constraints applied to it.

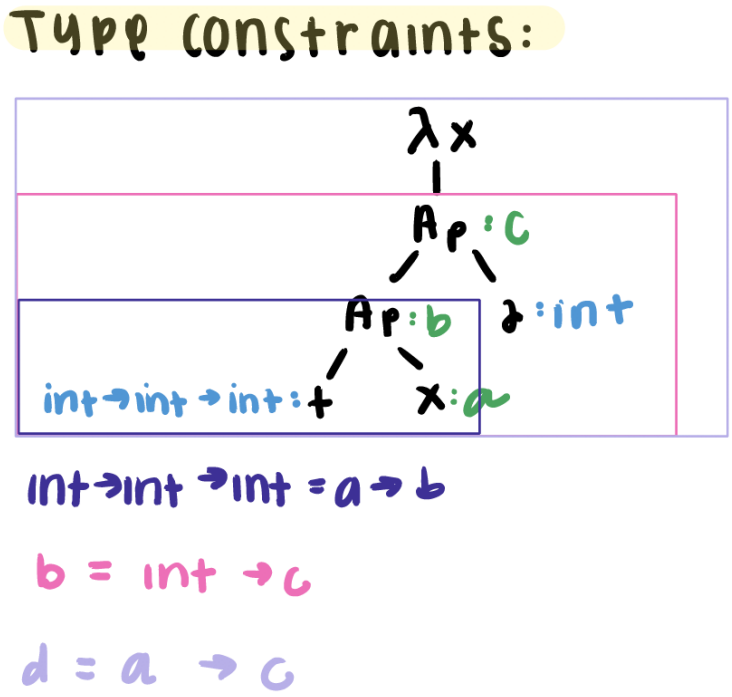
- An important note to make is that when you are determining the type constraints, a function is always an arrow type in the format:
- $function = parameter \rightarrow return$

2.3 Creating Constraints

After decorating the tree with each node's type, we need to come up with a list of constraints. For this example, the constraints are:

- $int \rightarrow int \rightarrow int = a \rightarrow b$
- $b = int \rightarrow c$
- $d = a \rightarrow c$

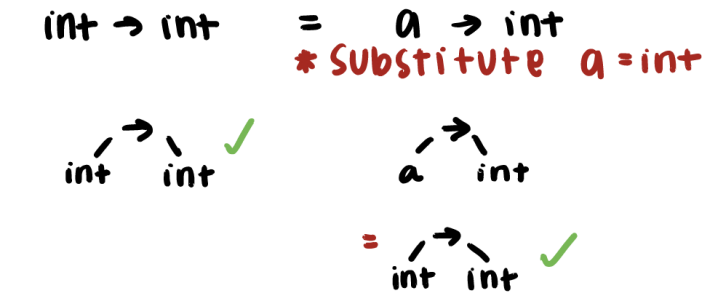
Here is an image of how these constraints look with respect to the tree:



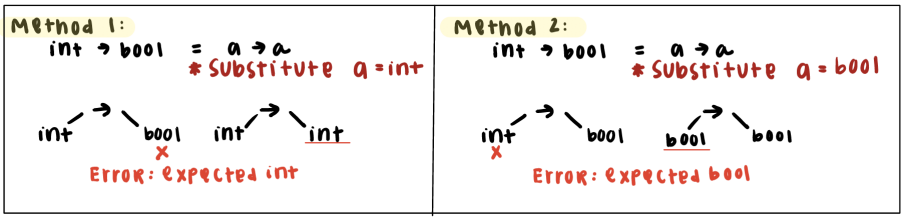
2.4 Passing Constraints to the Unifier

After we have finished generating the constraints we pass this information to the unifier to see if the type makes sense. In unification, you are making sure the types can be made the same using substitution.

- Suppose we have the constraint $int \to int = a \to int$; we need to substitute a for an int and see if the deduction in type is the same in both statements, which it is.
- Here is an example of the type checking during the unification (see next page):



- An example of when the unifer determines that the types do not match when using substitution is if you were trying to prove that $int \rightarrow bool = a \rightarrow a$. The reason that this fails is because if you assume $a = int$ then you will have to check that $int \rightarrow bool = int \rightarrow int$ which is not true since you are expecting a bool and you are providing an int. On the other hand, if you assume $a = bool$ then you will have to check that $int \rightarrow bool = bool \rightarrow bool$ which still does not hold since the function is expecting an int and you are providing a bool.
- As mentioned above, the error message will be different depending on how the unification is implemented.
- Here is an example of when the unification determines the type is not matched:



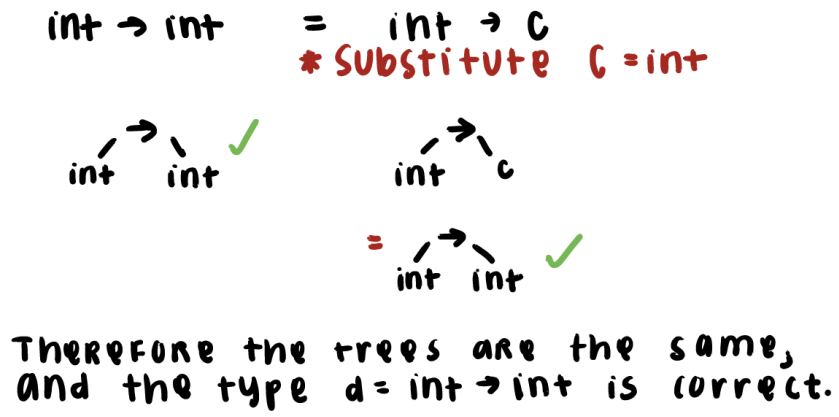
Example 9. Given the following constraints, use the unifier to determine if a type can be created.

- $int \to int \to int = a \to b$
- $b = int \to c$
- $d = a \to c$

Step 1 : With this information we can deduce the following information for the variable types:

- $a = int$
- $b = int \to int$
- $b = int \to c$
- $d = a \to c$ and since $a = int$ we know d is actually $d = int \to c$

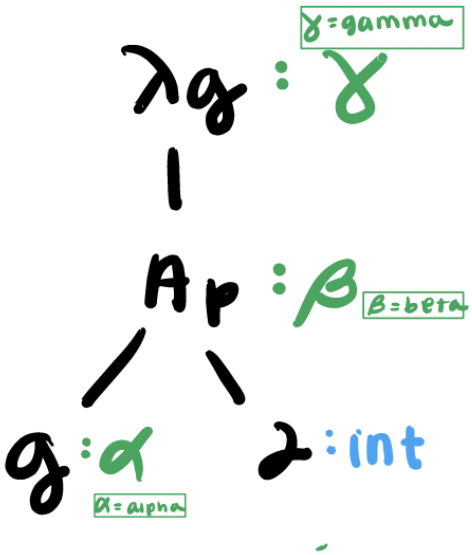
Step 2: Take the two different b statements: $b = int \to int$ and $b = int \to c$ and see if the unifier can use substitution to determine they are the same type:



Example 10. Determine the type of the following equation:

$$fun\ f\ g = g\ 2;$$

Step 1: Draw and decorate the corresponding tree using the information explained at the beginning of this document.



Step 2: Generate the constraints:

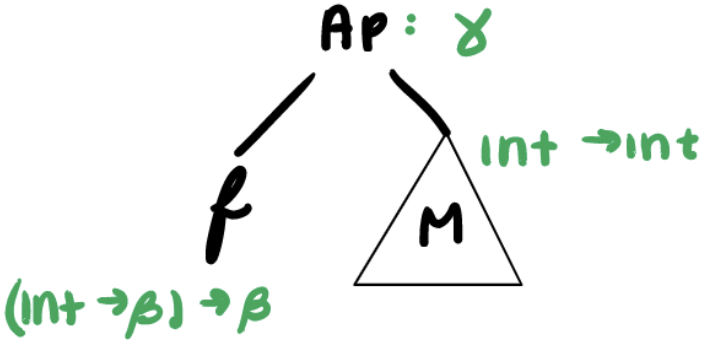
- $\alpha = int \to \beta$
- $\gamma = \alpha \to \beta$ which we can generate further to be $\gamma = (int \to \beta) \to \beta$ which is our final type.

To reiterate, the type for $fun\ f\ g\ = g\ 2;$ is $\gamma = (int \to \beta) \to \beta$.

Example 11. Determine the type of the following equation:

$$f(fn\ x\ = 2 + x)$$

Step 1: Decorate the tree



Step 2: Determine constraints

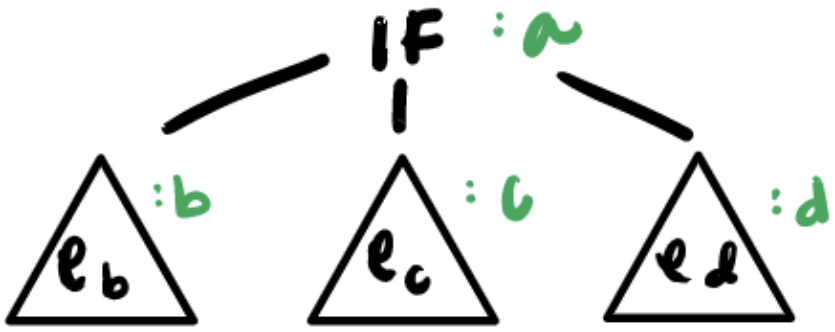
- $(int \to \beta) \to \beta = (int \to int) \to \gamma$
- $\beta = int$
- $\beta = \gamma$

Therefore: $\gamma = int$

Example 12. Determine the type of the following equation:

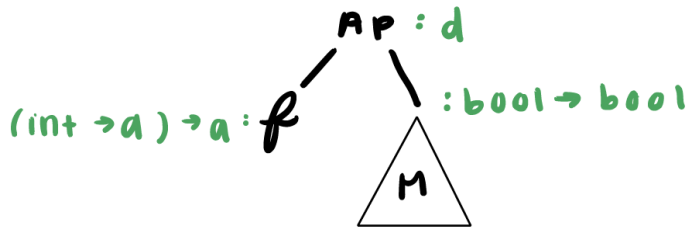
$$f(fn\ x =\ \ if\ x\ then\ false\ else\ true)$$

- Important Side Note regarding the type declarations for if statements:
- In an if statement, you have the general structure of "if bool then item else item". This means that the type of the expression after the "if" needs to be a bool and the types of the if and else items have to be the same type.
- Here is an example of a diagram for this if type structure:

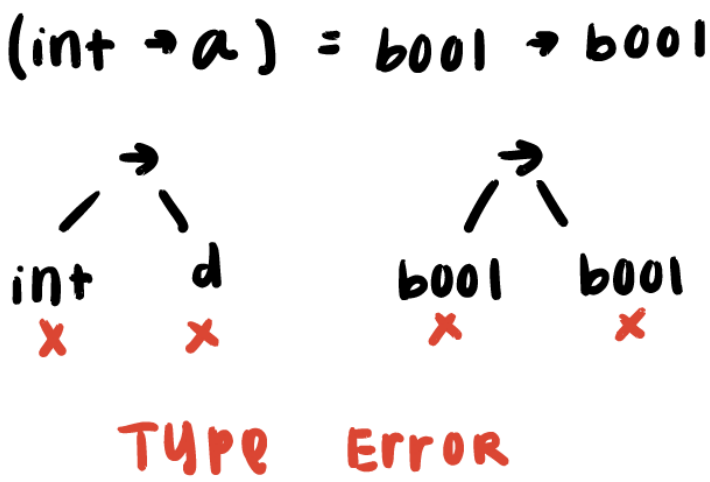


- In that diagram, b has to be a Boolean, and c and d have to be the same types, which matches the return type a.

Now back to the function example as a whole.
Step 1 is to decorate the tree with its types. For the decorations in this tree, it would follow the if diagram above, with $a = bool$, $b = bool$, $c = bool$, and $d = bool$. The reason these are all Booleans is because b has to be according to the if statement structure, and we know by the function declaration that c and d are bools because the if and else statments return true or false, which means the entire function returns a type bool.
Step 2: Pass the constraints to the unifier to make sure they are the same. Since these are all bools, the entire function has the type $bool \to bool$. However, the unifier determines that this is incorrect according to the function definition. The function states $int \to a$ and we determined the type would be $bool \to bool$ which cannot be the same.
Here is a diagram of the application:



and here is a diagram of the unifier giving a type error.



Therefore, there is an error in this function and a type cannot be determined.