

1 Type Inference

We begin today's discussion with Type Reconstruction, also known as Type Inference. Chapter 22 of the text presents the algorithm for constraint-based typing, which we discuss in class today. First off, we make the observation that in order to help with type checking, it is useful to make types for variables explicit in lambda abstractions. In interesting cases type inferencing is used to infer the type of variables, through the use of constraints. For instance, so far we have used our typing rules to prove type safety. In a more general sense, we've defined what terms are valid in our calculus. Yet, these rules are not very useful for type inferencing - something more is needed. For instance, our previous rules tell us nothing about what to do when a term may have more than one type.

That's where the notion of constraints comes in. A set of constraints should look like a set of bindings from variables to type variables. For instance, say our set of constraints is represented by the letter C. Then $C = \{ S = T \}$ where S and T are representations of types. For variables, we can have an empty set of constraints like so, where the constraints are contained in the curly braces in the judgement:

$$\frac{\Gamma(x) = t}{\Gamma \vdash x : t | \{ \}}$$

As for the lambda rule, it is important to notice that constraints are passed through the rules during the type derivation. One must figure out the constraints on the top first then pass the information down to the judgement. For instance, the rule for lambda now looks like:

$$\frac{\Gamma, x : t_1 \vdash e_2 : t_2 | C}{\Gamma \vdash \lambda x. e : t_1 \rightarrow t_2 | C}$$

The goal of constraint typing is to generate a set of constraints that will

enable us to better type the term in question. However, it is important to realize that given any term t , there might not always be a typing derivation for that term. There might be a set of constraints generated for that term, but that set may infact be unsatisfiable. Upon further inspection we see that the following is true about typing with constraints:

$$\forall e, \Gamma. \exists C, t. \Gamma \vdash e : t | C$$

Which says: For all expressions and environments, there exists some set of constraints and a type such that e is well-typed under Γ and satisfies the set of constraints C . However, there is no analog to this statment for the type system without constraints, because to say so would mean there was a type for any expression, well-typed or not. It is sensible to say the above because we are inspecting the structure of a term to produce constraints on the type of the term. We are trying to build a set of constraints based upon the information we can gather during the typing derivation.

The key to point to understand here is that there can be a set of constraints that is unsatisfiable - that is how you can have constraints on an ill-typed term. Yet, you can produce a potentially broken type in the above statement about constraint typing. This is because the information in C is incorrect and unsatisfiable. In a sense, a lot of the complexity is taken out of the typing system and put onto the shoulders of the constraints.

The rule for application carries with it a lot of information. First, we must notice that we are introducing fresh new type variables to represent the type of the application itself. This is represented in the form of X , the type of the application $e_1 e_2$:

$$\frac{\Gamma \vdash e_1 : t_1 | C_1 \quad \Gamma \vdash e_2 : t_2 | C_2}{\Gamma \vdash e_1 e_2 : X | \{t_1 = t_2 \rightarrow X\} \cup C_1 \cup C_2}$$

In the text, this rule carries much more information, like the fact that X cannot occur in either C_1 or C_2 , it must be fresh in this rule application. Furthermore, after X is generated, it must remain in the set of constraints for the rest of the derivation.

Finally, we present the extended rules as they appear in the text. The first

rule is for variables which carries with it an annotation, the zero subscript on the bar before the constraints. It indicates that no free variables are created in this rule.

$$\frac{\Gamma(x) = t}{\Gamma \vdash x : t|_0\{\}}$$

The rule for lambda carries more information as well. F represents the collection of free variables that has been previously created and needs to flow along with the rest of the typing derivation:

$$\frac{\Gamma, x : t_1 \vdash e_2 : t_2|_F C}{\Gamma \vdash \lambda x. e : t_1 \rightarrow t_2|_F C}$$

Finally, the rule for application adds X to the list of free variables already in the derivation: F_1 and F_2 :

$$\frac{\Gamma \vdash e_1 : t_1|_{F_1} C_1 \quad \Gamma \vdash e_2 : t_2|_{F_2} C_2}{\Gamma \vdash e_1 e_2 : X|_{F_1 \cup F_2 \cup \{x\}} \{t_1 = t_2 \rightarrow X\} \cup C_1 \cup C_2}$$