Lecture 4: Well-typed programs never go wrong!
Programming Languages (CSCI 3300), Fall 2014

Prof. Harley Eades (heades@gru.edu).

Many programming languages have types. For example, C# has the types bool, int, char, and many others. In fact, you can also construct new types by building classes. In a language like C# the type system is there to prevent the programmer from applying a method that expects something of one type to something of a different type. For example, trying to divide a string by a number is not allowed. This is perhaps the most obvious applications of types, but they amount to so much more.

We can think of a type as being a specification of what a program is supposed to do. For example, consider the type of the length function for lists:

```
length :: [a] -> Int
```

The type of length tells us that it takes a list of type `a` and returns an integer. So how many possible implementations are there for length? There are a lot! In fact, an infinite number of them. Here are a few of them:

```
length l = 0
length l = 1
length l = 2
length l = 3
```

However, due to the name these are not the ones we would expect. We would expect the implementation to be the following:

```
length :: [a] -> Int
length [] = 0
length (x:xs) = 1 + (length xs)
```

In some more advanced systems we could change the type to cut down on the number of possible implementations. Suppose p is a logical predicate that takes in a list and a number, and then is true when the length of the list is equal to the number. So for example, $p([1,2,3,4],4)$ is true, but $p([1,2,3,4,5],4)$ is false. Now if Haskell's type system was extended then we could change the type of the length function to not only return an integer, but to also return a proof that the number is the length of the list.

```
length :: (l : [a]) -> (n:Int,p l n)
```

If this were possible – and in some programming languages it is – then there would exist only one possible implementation of length and we would have a mathematical proof that this is true. **Thus, the type system can be used to enforce program correctness.** The more sophisticated the type system the better the specifications become, and thus, the more correct programs become. This is very profound. So if one is the programmer implementing the latter length function above, then when the programmer gets their implementation to type check, it is with mathematical certainty the correct program, because the type encodes the specification of what it is to be a length function using p. These style of types are called

dependent types because the types depend on other programs. For example, the return type of `length` above depends on the integer, and on p which can both be considered as programs.

In this lecture we start with a very basic type system for the λ-calculus. The kind of types we will be dealing with are called **simple types**. While they are simple they still have a very profound impact on the language. We introduce simple types by studying a type system designed by Kurt Gödel called system T.

1 System T

In this section we define Gödel's system T. This is just a fancy name, and do not worry, because this PL will simply be an extension of the λ-calculus.

1.1 Syntax of System T

The syntax of system T is as follows:

$$\begin{array}{rl} \text{(Types)} & T, A, B, C ::= \mathsf{Nat} \mid A \to B \\ \text{(Terms)} & t ::= x \mid 0 \mid \mathsf{suc}\, t \mid \lambda x : T.t \mid t_1\ t_2 \mid \mathsf{rec}\, t\ t_1\ t_2 \\ \text{(Contexts)} & \Gamma ::= \cdot \mid x : T \mid \Gamma_1, \Gamma_2 \end{array}$$

One can see that the syntax above is simply an extension of the syntax of the untyped λ-calculus. First, a new syntactic category for types as been added where Nat is the type for natural numbers, and $A \to B$ is the type of unary functions (λ-abstractions). Then the terms of the λ-calculus have been extended to include the natural number 0, the successor function $\mathsf{suc}\, t$, and the most interesting part recursion denoted $\mathsf{rec}\, t\ t_1\ t_2$. Finally, we add a second new syntactic category called contexts. These should be considered as lists of pairs of variables and types denoted $x : T$. These will be used to keep track of the types of free variables during type checking – we introduce type checking below.

Parsing Conventions. All of the parsing conversions we have been using still stand, but there is one new one which is that the function type associates to the right. That is,

$$T_1 \to T_2 \to \cdots \to T_i \equiv (T_1 \to (T_2 \to (\cdots (T_{i-1} \to T_i) \cdots)))$$

Note that this is opposite to application.

1.2 Types in System T

Perhaps the biggest addition to system T is the notion of a type. The most obvious application of types is to insure that the correct data is given to a function when applying it, however, types provide so much more, but first we must understand the basics.

The function type $A \to B$ means that a function with this type only accepts arguments whose type is A, and once given an argument of type A the function will produce something of type B. These are equivalent to function types in Haskell. Now in system T the λ-abstractions are actually annotated with the type of the argument. For example, $\lambda x : \mathsf{Nat}.x$ is the identity function that takes in only a natural number, and then returns the same number. Thus, its type is $\mathsf{Nat} \to \mathsf{Nat}$. If one was to apply this function to anything else then the application would fail to type check, but what is type checking?

Type checking corresponds to an algorithm that takes as input a term, and a list of types for each free variable of the input term, and then computes the type of the term. We denote this by $\Gamma \vdash t : T$ where Γ is the list of free variables with their types called a **typing context**, t is the input term, T is the proposed type. Then we **check** to see if t has type T by constructing a derivation using the type checking inference

rules. This is similar to how we show that a term reduces to another using the reduction rules. The type checking rules are as follows:

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T}\ \text{T\_Var} \qquad \frac{\Gamma, x : T_1 \vdash t : T_2}{\Gamma \vdash \lambda x : T_1.t : T_1 \to T_2}\ \text{T\_Lam} \qquad \frac{\Gamma \vdash t_1 : T_1 \to T_2 \quad \Gamma \vdash t_2 : T_1}{\Gamma \vdash t_1\, t_2 : T_2}\ \text{T\_App}$$

$$\frac{}{\Gamma \vdash 0 : \mathsf{Nat}}\ \text{T\_Zero} \qquad \frac{\Gamma \vdash t : \mathsf{Nat}}{\Gamma \vdash \mathsf{suc}\, t : \mathsf{Nat}}\ \text{T\_Suc} \qquad \frac{\begin{array}{c}\Gamma \vdash t : \mathsf{Nat} \\ \Gamma \vdash t_1 : T \quad \Gamma \vdash t_2 : T \to \mathsf{Nat} \to T\end{array}}{\Gamma \vdash \mathsf{rec}\, t\, t_1\, t_2 : T}\ \text{T\_Rec}$$

Notice that there is a typing rule per term (or program) of system T. The typing rule T_Var says that a variable x has type T if the typing context says it does. Then the rule T_Lam says that if t has type T_2 in a context where x has type T_1 – here x is the argument to the function – then $\lambda x : T_1.t$ has type $T_1 \to T_2$. If t_1 has type $T_1 \to T_2$ – so t_1 is a function – and t_2 has type T_1, then the application $t_1\, t_2$ has type T_2. So the rule T_App captures the fact that application really is function application. Note that this limits which terms can be arguments to functions which is very different from the untyped λ-calculus. Now the rule T_Zero states that no matter what 0 is a Nat. Similarly, if we know t is a natural number, then we should know that $\mathsf{suc}\, t$ is, and this is exactly what the rule T_Suc gives us.

Probably the most interesting and complex rule is the one for the recursor $\mathsf{rec}\, t\, t_1\, t_2$. Each piece of the recursor should be interpreted as follows:

- The term t is a natural number called the counter,
- The term t_1 is the base case (when t is zero), and
- The term t_2 is the step case.

So the typing rule T_Rec says that if t is a natural number, that is, of type Nat, the base case t_1 is of type T, and the step case t_2 is of type $T \to \mathsf{Nat} \to T$, then the term $\mathsf{rec}\, t\, t_1\, t_2$ has type T. One should think of the recursor as computing something of type T using recursion.

Now we consider several examples of type checking terms:

- Determine if the term $\lambda x : \mathsf{Nat}.x$ has type $\mathsf{Nat} \to \mathsf{Nat}$. First, notice that there are no free variables in $\lambda x : \mathsf{Nat}.x$ and so the context will start out empty. The typing derivation is as follows:

$$\dfrac{\dfrac{\dfrac{x : \mathsf{Nat} \in x : \mathsf{Nat}}{x : \mathsf{Nat} \vdash x : \mathsf{Nat}}\ \text{T\_Var}}{\cdot \vdash \lambda x : \mathsf{Nat}.x : \mathsf{Nat} \to \mathsf{Nat}}\ \text{T\_Lam}}$$

- Determine if the term $\lambda x : \mathsf{Nat} \to \mathsf{Nat}.\lambda y : \mathsf{Nat}.x\, y$ has type $\mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat}$:

$$\dfrac{\dfrac{\dfrac{\dfrac{x : \mathsf{Nat} \to \mathsf{Nat} \in x : \mathsf{Nat} \to \mathsf{Nat}, y : \mathsf{Nat}}{x : \mathsf{Nat} \to \mathsf{Nat}, y : \mathsf{Nat} \vdash x : \mathsf{Nat} \to \mathsf{Nat}}\ \text{T\_Var} \qquad \dfrac{y : \mathsf{Nat} \in x : \mathsf{Nat} \to \mathsf{Nat}, y : \mathsf{Nat}}{x : \mathsf{Nat} \to \mathsf{Nat}, y : \mathsf{Nat} \vdash y : \mathsf{Nat}}\ \text{T\_Var}}{x : \mathsf{Nat} \to \mathsf{Nat}, y : \mathsf{Nat} \vdash x\, y : \mathsf{Nat}}\ \text{T\_App}}{x : \mathsf{Nat} \to \mathsf{Nat} \vdash \lambda y : \mathsf{Nat}.x\, y : \mathsf{Nat} \to \mathsf{Nat}}\ \text{T\_Lam}}{\cdot \vdash \lambda x : \mathsf{Nat} \to \mathsf{Nat}.\lambda y : \mathsf{Nat}.x\, y : \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat}}\ \text{T\_Lam}$$

- Determine if the term $\lambda x : \mathsf{Nat}.x\, y$ has type $\mathsf{Nat} \to \mathsf{Nat}$ when y has type Nat. Notice here y is free so we must start out with y in the context:

$$\dfrac{\dfrac{y : \mathsf{Nat}, x : \mathsf{Nat} \vdash x : \mathsf{Nat} \to \mathsf{Nat} \qquad \dfrac{y : \mathsf{Nat} \in y : \mathsf{Nat}, x : \mathsf{Nat}}{y : \mathsf{Nat}, x : \mathsf{Nat} \vdash y : \mathsf{Nat}}\ \text{T\_Var}}{y : \mathsf{Nat}, x : \mathsf{Nat} \vdash x\, y : \mathsf{Nat}}\ \text{T\_App}}{y : \mathsf{Nat} \vdash \lambda x : \mathsf{Nat}.x\, y : \mathsf{Nat} \to \mathsf{Nat}}\ \text{T\_Lam}$$

The previous derivation fails, because x has type Nat, but we needed it to have type $\mathsf{Nat} \to \mathsf{Nat}$. Thus, the term $\lambda x : \mathsf{Nat}.x\ y$ does not have type $\mathsf{Nat} \to \mathsf{Nat}$. We call this a type error.

1.3 Computing in System T

$$\frac{}{(\lambda x : T.t)\ t' \rightsquigarrow [t'/x]t}\ \text{E\_Beta} \qquad \frac{}{\mathsf{rec}\ 0\ t_1\ t_2 \rightsquigarrow t_1}\ \text{E\_RecBase}$$

$$\frac{}{\mathsf{rec}\ (\mathsf{suc}\ t)\ t_1\ t_2 \rightsquigarrow (t_2\ (\mathsf{rec}\ t\ t_1\ t_2))\ t}\ \text{E\_RecStep} \qquad \frac{t \rightsquigarrow t'}{\lambda x : T.t \rightsquigarrow \lambda x : T.t'}\ \text{E\_Lam}$$

$$\frac{t_1 \rightsquigarrow t_1'}{t_1\ t_2 \rightsquigarrow t_1'\ t_2}\ \text{E\_App1} \qquad \frac{t_2 \rightsquigarrow t_2'}{t_1\ t_2 \rightsquigarrow t_1\ t_2'}\ \text{E\_App2} \qquad \frac{t \rightsquigarrow t'}{\mathsf{suc}\ t \rightsquigarrow \mathsf{suc}\ t'}\ \text{E\_Suc}$$

$$\frac{t \rightsquigarrow t'}{\mathsf{rec}\ t\ t_1\ t_2 \rightsquigarrow \mathsf{rec}\ t'\ t_1\ t_2}\ \text{E\_Rec1} \qquad \frac{t_1 \rightsquigarrow t_1'}{\mathsf{rec}\ t\ t_1\ t_2 \rightsquigarrow \mathsf{rec}\ t\ t_1'\ t_2}\ \text{E\_Rec2} \qquad \frac{t_2 \rightsquigarrow t_2'}{\mathsf{rec}\ t\ t_1\ t_2 \rightsquigarrow \mathsf{rec}\ t\ t_1\ t_2'}\ \text{E\_Rec3}$$