Lecture 4: Simply-Typed λ-Calculus

Ankush Das

January 30, 2024

1 Introduction

Today, we will study how types can prevent programmers from making mistakes while writing programs. A type checker is simply a validity checker for a program to make sure we do not accidentally execute invalid programs.

2 The LL1 Type Checker

Let's return to the simple arithmetic and boolean expression language introduced in Lecture 1 and Homework 1. The syntax is written as

$$\text{Expressions} \quad e ::= e + e \mid \mathsf{if}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e \mid \overline{n} \mid \mathsf{true} \mid \mathsf{false}$$

Since we have only two kinds of expressions in our language, we need two types

$$\text{Types} \quad \tau ::= \mathsf{int} \mid \mathsf{bool}$$

Now, we define the rules of the type system. We use a simple judgment written as $e : \tau$ meaning that expression e has type τ. Again, the principle for defining the type system is simple. We take every expression defined in the syntax and we define the rule for typing that expression. The rules are defined as

$$\frac{e_1 : \mathsf{int} \qquad e_2 : \mathsf{int}}{e_1 + e_2 : \mathsf{int}}\ \text{T-Add} \qquad \frac{e : \mathsf{bool} \qquad e_1 : \tau \qquad e_2 : \tau}{\mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : \tau}\ \text{T-If} \qquad \frac{}{\overline{n} : \mathsf{int}}\ \text{T-Num}$$

$$\frac{}{\mathsf{true} : \mathsf{bool}}\ \text{T-TT} \qquad \frac{}{\mathsf{false} : \mathsf{bool}}\ \text{T-FF}$$

The T-Add rule defines that $e_1 + e_2$ can only have type int and for that e_1 and e_2 must have type int. The T-If rule dictates that e must have type bool and e_1 and e_2 must have the same arbitrary type τ and then the 'if' expression has the same type τ. The remaining rules just describe how to type values.

An interesting thing to note in this language is that there are no variables. That's because there is no way to introduce variables. Now, we can either introduce functions (like λ-calculus) or we can introduce a let expression: $\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$, meaning x will have value that e_1 evaluates to in the expression e_2.

How do we type this expression?

$$\frac{e_1 : \tau' \qquad e_2 : \tau?}{\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau}\ \text{T-Let}$$

This doesn't seem entirely correct. The expression e_2 can refer to variable x but that is not specified as a premise when typing e_2.

Thus, we need to introduce a typing context, which we call Γ that tracks the types of all the variables in the context (e.g., $\Gamma = \{x_1 : \tau_1, x_2 : \tau_2, \ldots\}$). We mandate that all variables in Γ are distinct. We slightly modify the typing judgment as $\Gamma \vdash e : \tau$ meaning that expression e has type τ in the presence of context Γ, i.e., in the presence of $x_1 : \tau_1, x_2 : \tau_2, \ldots$. Thus, we re-write the T-Let rule

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma, x : \tau_1 \vdash e_2 : \tau}{\Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau}\ \text{T-Let}$$

3 A Brief Introduction to Logic

In this section, we will study one of the coolest results in PL theory: the Curry-Howard isomorphism. We will see how programming languages are closely connected to logic. To understand this cool result, let's see intuitionistic logic and how it closely connects to λ-calculus. Since λ-calculus is all about functions, we look at the introduction and elimination rules for implication.

$$\frac{\Gamma, \alpha \vdash \beta}{\Gamma \vdash \alpha \to \beta} \to\text{I} \qquad \frac{\Gamma \vdash \alpha \to \beta \qquad \Gamma \vdash \alpha}{\Gamma \vdash \beta} \to\text{E}$$

Now, all we will do is add terms to the above rules.

$$\frac{\Gamma, x : \alpha \vdash e : \beta}{\Gamma \vdash \lambda x.e : \alpha \to \beta} \to\text{I} \qquad \frac{\Gamma \vdash e_1 : \alpha \to \beta \qquad \Gamma \vdash e_2 : \alpha}{\Gamma \vdash e_1\ e_2 : \beta} \to\text{E}$$

We have one last rule remaining, typing variables. This is derived from the id rule in intuitionistic logic.

$$\frac{}{\Gamma, \alpha \vdash \alpha}\ \mathsf{id} \qquad \frac{}{\Gamma, x : \alpha \vdash x : \alpha}\ \textsc{Var}$$

That's it! That's all there is to the type system of the λ-calculus.

With this, I'd like to remind readers that defining a programming language now requires 3 components:

- **Syntax**: how to write programs
- **Type System**: what programs are valid for execution, and
- **Semantics**: how to execute valid programs

We conclude this lecture by proving type safety of λ-calculus. We already saw the progress theorem, we will now define the preservation theorem.

Theorem 1 (Preservation)**.** *For all closed well-typed expressions e, i.e., $\cdot \vdash e : \tau$, if $e \mapsto e'$, then $\cdot \vdash e' : \tau$.*

This theorem states that if a well-typed expression takes a step, the new expression has the same type as the original expression. Also note that expression e is closed, since the context to type e is empty. Now, let's go about proving this theorem.

$$\frac{}{\lambda x.e\ \mathsf{value}}\ \lambda\text{-V} \qquad \frac{e_1 \mapsto e_1'}{e_1\ e_2 \mapsto e_1'\ e_2}\ \textsc{App-L} \qquad \frac{e_1\ \mathsf{value} \qquad e_2 \mapsto e_2'}{e_1\ e_2 \mapsto e_1\ e_2'}\ \textsc{App-R}$$

$$\frac{e'\ \mathsf{value}}{(\lambda x.e)\ e' \mapsto [e'/x]e}\ \textsc{App-S}$$

Again, we prove by induction on the derivation of $e \mapsto e'$. Recall the rules of the semantics. There are three cases, all for function application since λ-expressions and variables cannot take a step. Hence, all the cases are when $e = e_1\ e_2$ and $\cdot \vdash e : \tau$, which implies

$$\frac{\cdot \vdash e_1 : \alpha \to \tau \qquad \cdot \vdash e_2 : \alpha}{\cdot \vdash e_1\ e_2 : \tau} \to\text{E}$$

- Case when

  $$\frac{e_1 \mapsto e_1'}{e_1\ e_2 \mapsto e_1'\ e_2}\ \textsc{App-L}$$

  In this case, we appeal to the inductive hypothesis for $e_1 \mapsto e_1'$. We note that $\cdot \vdash e_1 : \alpha \to \tau$ and conclude $\cdot \vdash e_1' : \alpha$. This means we can apply the $\to$E rule again.

  $$\frac{\cdot \vdash e_1' : \alpha \to \tau \qquad \cdot \vdash e_2 : \alpha}{\cdot \vdash e_1'\ e_2 : \tau} \to\text{E}$$

  Hence, $\cdot \vdash e' : \tau$ since $e' = e_1'\ e_2$.

- Case when

$$\frac{e_1 \text{ value} \qquad e_2 \mapsto e_2'}{e_1\ e_2 \mapsto e_1\ e_2'}\ \textsc{App-R}$$

We appeal to the inductive hypothesis for $e_2 \mapsto e_2'$ and since $\cdot \vdash e_2 : \alpha$, we conclude $\cdot \vdash e_2' : \alpha$. Again, we apply the $\to$E rule.

$$\frac{\cdot \vdash e_1 : \alpha \to \tau \qquad \cdot \vdash e_2' : \alpha}{\cdot \vdash e_1\ e_2' : \tau} \to\text{E}$$

Again, $\cdot \vdash e' : \tau$ because $e' = e_1\ e_2'$.

- Case when

$$\frac{e' \text{ value}}{(\lambda x.e)\ e' \mapsto [e'/x]e}\ \textsc{App-S}$$

Let's consider the typing in this situation.

$$\frac{\dfrac{x : \alpha \vdash e : \tau}{\cdot \vdash \lambda x.e : \alpha \to \tau} \to\text{I} \qquad \cdot \vdash e' : \alpha}{(\lambda x.e)\ e' \mapsto [e'/x]e} \to\text{E}$$

Now, we're stuck. Our goal is to prove that $\cdot \vdash [e'/x]e : \tau$ but we don't know how to prove this lemma because we have not done proofs on terms with substitution.

Lemma 1 (Substitution)**.** *If* $\Gamma \vdash e : \tau$ *and* $\Gamma', x : \tau \vdash e' : \tau'$ *then* $\Gamma, \Gamma' \vdash [e/x]e' : \tau$.

Proof. Just like every other proof, this proof will also occur by induction on the typing judgment $\Gamma', x : \tau \vdash e' : \tau$. This is an exercise for the reader. □

Now, that we have a proof of the substitution lemma, we can use that above as follows: we know that $\cdot \vdash e' : \alpha$ (second premise), and we know that $x : \alpha \vdash e : \tau$. Using the two in the substitution lemma, we get that $\cdot \vdash [e'/x]e : \tau$.