Tutorial 7

Computer Language Processing and Compiler Design
(COMP 4901U)

November 1, 2021

Exercise (1): Simple Typing

Consider the following typing rules:

$$\textsc{Lit}\ \frac{n \text{ is an integer literal}}{n : \mathsf{Int}} \qquad \textsc{Add}\ \frac{t_1 : \mathsf{Int} \qquad t_2 : \mathsf{Int}}{t_1 + t_2 : \mathsf{Int}} \qquad \textsc{Mul}\ \frac{t_1 : \mathsf{Int} \qquad t_2 : \mathsf{Int}}{t_1 \ * \ t_2 : \mathsf{Int}}$$

$$\textsc{True}\ \frac{}{\textbf{true} : \mathsf{Bool}} \qquad \textsc{Not}\ \frac{t : \mathsf{Bool}}{!t : \mathsf{Bool}} \qquad \textsc{EqInt}\ \frac{t_1 : \mathsf{Int} \qquad t_2 : \mathsf{Int}}{t_1 == t_2 : \mathsf{Bool}} \qquad \textsc{EqBool}\ \frac{t_1 : \mathsf{Bool} \qquad t_2 : \mathsf{Bool}}{t_1 == t_2 : \mathsf{Bool}}$$

Question 1

Write down a typing derivations for

$$3 + (5 * 6) : \mathsf{Int}$$

and for

$$(2 == 3) == !\textbf{true} : \mathsf{Bool}$$

Question 2

We say that a term t is ***well-typed*** if there exists a type T and a typing derivation which concludes $t : T$. Note that the derivations you provided above are indeed constructive proofs that $3 + (5 * 6)$ and $(2 == 3) == !\textbf{true}$ are well-typed.

In this question, your goal is to formally prove that:

- $1 == \textsf{true}$ is *not* well-typed. That is, prove that there exist *no* T such that there exists a derivation of $1 == \textsf{true} : T$.

- Every subterm of a well-typed term is well-typed.

- Any given term t can only be assigned a unique type T such that $t : T$ holds (i.e., is derivable).

Question 3

Provide the rules of operational semantics for the language, using the standard interpretation of $+$, $*$, $!$, and $==$. We assume that the values expressions evaluate to are integer literals n, the true literal **true**, and its negation **!true**.

We give two example rule below, and you have to come up with the others:

$$\textsc{E-Add1}\ \frac{\mathit{value}(n_3) = \mathit{value}(n_1) + \mathit{value}(n_2)}{n_1 + n_2 \rightsquigarrow n_3} \qquad \textsc{E-Add2}\ \frac{t_1 \rightsquigarrow t_1'}{t_1 + t_2 \rightsquigarrow t_1' + t_2} \qquad \ldots$$

Make sure that your rules are ***deterministic*** — that is, for any given term t, there is at most one t' such that $t \rightsquigarrow t'$ can be derived and there is at most one derivation of $t \rightsquigarrow t'$.

Question 4

Prove that no well-typed term "*gets stuck*". That is, prove that if t is well-typed, then either t is an integer literal n, or t is **true** or **!true**, or there exists a t' such that $t \rightsquigarrow t'$. This property is known as "***progress***".

Then, prove that if t is well-typed and $t \rightsquigarrow t'$ for some t', then t' is also well-typed. This property is known as "***preservation***".

Question 5

We want to extend this simple language and its type system to support expressions containing nested **val** bindings and lambda expressions, like in Scala. For instance, given term $t = (\textbf{val}\ x : \textsf{Int} = 4;\ \textbf{val}\ y : \textsf{Int} = x + x;\ x * y)$, we should be able to derive $t : \textsf{Int}$. Similarly, we should type check the lambda expression $((x : \textsf{Int}) \Rightarrow x + 1)$ as of type $(\textsf{Int}) \Rightarrow \textsf{Int}$.

To do this, we have to *generalize* our typing rules by adding a *typing context* Γ to them, whose syntax is as follows:

$$\Gamma \ ::= \ \varepsilon \ | \ \Gamma \cdot (x : T)$$

We define the operation written $(x, T) \in \Gamma$ for retrieving a binding (x, T) in a context Γ as follows:

$$\textsc{Gamma1}\ \frac{}{(x, T) \in \Gamma \cdot (x, T)} \qquad \textsc{Gamma2}\ \frac{x \neq y \qquad (x, T) \in \Gamma}{(x, T) \in \Gamma \cdot (y, T')}$$

For example, we can show that we have $(x, T) \in \varepsilon \cdot (y, S) \cdot (x, T) \cdot (z, U)$ and that we have $(x, T) \notin \varepsilon$.

Note: this definition implements *shadowing* semantics for bindings; indeed, we have $(x, T) \in \varepsilon \cdot (x, S) \cdot (x, T)$, where we can see that the second binding of x *shadows* the first one.

Using these definitions, write the new typing rules of our language extended with the "**val** $x = t_1;\ t_2$" and "$(x_1 : T_1, \ldots, x_n : T_n) \Rightarrow t$" syntax forms. By convention, your typing rules should use the syntax $\Gamma \vdash t : T$.

Question 6

In this extended system, are there terms that can be given more than one type? If so, give at least two examples of this.

Exercise (2): Call-By-Need

Consider a simple programming language with integer arithmetic, boolean expressions and user-defined functions.

$$\begin{aligned} T \ &::= \ \mathsf{Int} \mid \mathsf{Bool} \mid (T, \ldots, T) \Rightarrow T \\ t \ &::= \ \textbf{true} \mid \textbf{false} \mid c \mid t == t \mid t + t \mid t \mathbin{\&\&} t \\ &\quad\ \mid \textbf{if}\ t\ \textbf{then}\ t\ \textbf{else}\ t \mid f(t, ..., t) \mid x \end{aligned}$$

Where c represents integer literals, $==$ represents equality (between integers, as well as between booleans), $+$ represents the usual integer addition and $\&\&$ represents conjunction. The meta-variable f refers to names of user-defined function and x refers to the names of variables.

Assume you have a fixed environment e which contains information about user-defined functions (i.e. the function parameters, their types, the function body and the result type).

$$e = \{\ f_1 \mapsto ([x_1,\ \ldots,\ x_n],\ [T_1,\ \ldots,\ T_n],\ t_{\mathsf{body}},\ T_{\mathsf{result}});\ f_2 \mapsto (\ldots);\ \ldots\ \}$$

Notation: $e(f_1)$ denotes the information associated with f_1, i.e., in the example above, $([x_1,\ \ldots,\ x_n],\ [T_1,\ \ldots,\ T_n],\ t_{\mathsf{body}},\ T_{\mathsf{result}})$.

Question 1

Write down the natural typing rules for this language.

Question 2

Inductively define (i.e., using inference rules) the ***substitution*** operation for your terms, which replaces every free occurrence of a variable in an expression by an expression without free variables.

Notation: we will write $t_1[x := t_2] \to t_3$ to denote that the substitution, in t_1, of every occurrence of x by t_2 results in term t_3.

Question 3

Prove that substitution preserves the type of an expression, given that the variable and the expression have the same type.

Question 4

Write the operational semantics rules for the language, assuming ***call-by-name*** semantics for function calls.

In call-by-name semantics, the arguments of a function are not evaluated before the call. In your operational semantics, parameters in the function body are to merely be substituted by the corresponding unevaluated argument expression.

Question 5

Can the soundness proofs seen in the lecture be easily adapted to this new semantics? What changes do we need to make?