Tutorial 7 (Solutions)

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

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

Solution

$$\dfrac{\dfrac{}{3 : \mathsf{Int}}\ \textsc{Lit} \qquad \dfrac{\dfrac{}{5 : \mathsf{Int}}\ \textsc{Lit} \qquad \dfrac{}{6 : \mathsf{Int}}\ \textsc{Lit}}{5 * 6 : \mathsf{Int}}\ \textsc{Mul}}{3 + (5 * 6) : \mathsf{Int}}\ \textsc{Add}$$

$$\dfrac{\dfrac{\dfrac{}{2 : \mathsf{Int}}\,\textsc{Lit} \qquad \dfrac{}{3 : \mathsf{Int}}\,\textsc{Lit}}{2 == 3 : \mathsf{Bool}}\,\textsc{EqInt} \qquad \dfrac{\dfrac{}{\textbf{true} : \mathsf{Bool}}\,\textsc{True}}{!\textbf{true} : \mathsf{Bool}}\,\textsc{Not}}{(2 == 3) == !\textbf{true} : \mathsf{Bool}}\,\textsc{EqBool}$$

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:

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

Solution

- Notice that proving the *non-existence* of a derivation is harder than proving its existence, as the latter simply requires an example.

  Here, we need to prove the non-existence statement at hand by contradiction: suppose there exists a derivation of $1 == \textbf{true}$. The last rule being applied must either be EqInt or EqBool, since these are the only rules that can assign types to equality terms. By looking at the premises of these rules, we can see that we must either have (a) both $1 : \mathsf{Int}$ *and* $\textbf{true} : \mathsf{Int}$; or (b) both $1 : \mathsf{Bool}$ *and* $\textbf{true} : \mathsf{Bool}$. But we can show that $\textbf{true} : \mathsf{Int}$ cannot be derived, since there are no rules that can assign a type other than Bool to term $\textbf{true}$, which rules out (a). Similarly, we can show that $1 : \mathsf{Bool}$ cannot be derived, ruling out (b), and we have a contradiction.

- We prove this by induction on typing derivations and by case analysis on the last rule being applied to derive $t : T$, which gives us subderivations for the immediate subterms of t.

- We need to prove that assuming $t : T_1$ and $t : T_2$ can be derived, then we must have $T_1 = T_2$. This can be done, again, by induction on typing derivations. At each step, we show that assuming the property for all subderivations, the same rule must have applied last, and therefore T_1 is the same as T_2.

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:

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

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 $t \rightsquigarrow t'$.

Solution

$$\textsc{E-Add1}\ \frac{\mathit{value}(n_3) = \mathit{value}(n_1) + \mathit{value}(n_2)}{n_1 + n_2 \rightsquigarrow n_3} \qquad \textsc{E-Add2}\ \frac{t_1 \rightsquigarrow t_1'}{t_1 + t_2 \rightsquigarrow t_1' + t_2} \qquad \textsc{E-Add3}\ \frac{t \rightsquigarrow t'}{n + t \rightsquigarrow n + t'}$$

[Rules for Mul similar to Add.]

$$\textsc{E-Not1}\ \frac{}{!!\textsf{true} \rightsquigarrow \textsf{true}} \qquad \textsc{E-Not2}\ \frac{t \rightsquigarrow t'}{!t \rightsquigarrow !t'}$$

$$\textsc{E-EqInt1}\ \frac{}{n == n \rightsquigarrow \textsf{true}} \qquad \textsc{E-EqInt2}\ \frac{n_1 \neq n_2}{n_1 == n_2 \rightsquigarrow !\textsf{true}} \qquad \textsc{E-EqInt3}\ \frac{t_1 \rightsquigarrow t_1'}{t_1 == t_2 \rightsquigarrow t_1' == t_2}$$

$$\textsc{E-EqInt4}\ \frac{t_2 \rightsquigarrow t_2'}{n == t_2 \rightsquigarrow n == t_2'}$$

[Rules for EqBool similar to EqInt.]

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***".

Solution

We prove both of these facts by straightforward induction on typing derivations. For preservation, we prove the stronger statement that "if $t : T$ and $t \rightsquigarrow t'$, then $t' : T$" (i.e., the type of t' is the same as that of t').

- Case Lit: By the shape of the rule, we know $t = n$ and $t : \mathsf{Int}$.
  Progress: Immediate, since t is an integer literal n.
  Preservation: Immediate, since there are no rules such that $n \rightsquigarrow t$ for some t.

- Case Add: By the shape of the rule, we know $t = t_1 + t_2$ where $t_1 : \mathsf{Int}$ and $t_2 : \mathsf{Int}$ and $t : \mathsf{Int}$.
  Progress: Note that we can't have any of $t_1, t_2 \in \{\textsf{true};\ !\textsf{true}\}$ as

that would make t ill-typed. If $t_1 = n_1$ and $t_1 = n_2$ for some n_1 and n_2, then rule E-ADD1 applies. Otherwise, if t_1 is not a value, by induction we have $t_1 \rightsquigarrow t_1'$ and rule E-ADD2 applies. Otherwise, if $t_1 = n_1$ but t_2 is not a value, by induction we have $t_2 \rightsquigarrow t_2'$ and rule E-ADD3 applies. This analysis covers all possible cases.
Preservation: Assuming E-ADD1 is the evaluation rule that is used, then the result n_3, an integer literal, has the same Int type as t. Assuming E-ADD2 is the evaluation rule that is used, then by induction we know that $t_1' : \mathsf{Int}$ and so the result $t_1' + t_2 : \mathsf{Int}$. Similarly for E-ADD3.

- Case ...etc. ...

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 $t = (\textbf{val } x : \mathsf{Int} = 4;\ \textbf{val } y : \mathsf{Int} = x + x;\ x * y)$, we should be able to derive $t : \mathsf{Int}$. Similarly, we should type check the lambda expression $((x : \mathsf{Int}) \Rightarrow x + 1)$ as of type $(\mathsf{Int}) \Rightarrow \mathsf{Int}$.

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

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

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 "$\textbf{val } x = t_1;\ t_2$" and "$(x_1 : T_1,\, \dots,\, x_n : T_n) \Rightarrow t$" syntax forms. By convention, your typing rules should use the syntax $\Gamma \vdash t : T$.

Solution

$$\textsc{Var}\ \frac{(x : T) \in \Gamma}{\Gamma \vdash x : T} \qquad \textsc{Val}\ \frac{\Gamma \vdash t_1 : T_1 \qquad \Gamma \cdot (x : T_1) \vdash t_2 : T_2}{\Gamma \vdash (\textsf{val}\ x = t_1;\ t_2) : T_2}$$

$$\textsc{Lam}\ \frac{\Gamma \cdot (x_1 : T_1) \cdot \ldots \cdot (x_n : T_n) \vdash t : T}{\Gamma \vdash ((x_1 : T_1, \ldots, x_n : T_n) \Rightarrow t) : (T_1, \ldots, T_n) \Rightarrow T}$$

[The other rules are as before, except that we add $\Gamma \vdash \ldots$ in front.]

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.

Solution

Yes, because the semantics of terms changes depending on the given typing context:

- x has type $\textsf{Int}$ in context $\varepsilon \cdot (x, \textsf{Int})$ and type $\textsf{Bool}$ in context $\varepsilon \cdot (x, \textsf{Bool})$.
- Similarly, we have $\varepsilon \cdot (x, \textsf{Int}) \cdot (y, \textsf{Int}) \vdash x == y : \textsf{Int}$ and $\varepsilon \cdot (x, \textsf{Bool}) \cdot (y, \textsf{Bool}) \vdash x == y : \textsf{Bool}$.

On the other hand, *closed* terms (i.e., those which contain no free variables) are uniquely typed, regardless of the context.

Exercise (2): Call-By-Need

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

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

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.

Solution

We start from context $\Gamma_0 = \{(f, T) \mid e(f) = ([x_1, \ldots], [T_1, \ldots], t, T)\}$.

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

$$\textsc{EqInt}\quad \frac{\Gamma \vdash t_1 : \mathsf{Int} \qquad \Gamma \vdash t_2 : \mathsf{Int}}{\Gamma \vdash t_1 == t_2 : \mathsf{Bool}} \qquad \textsc{EqBool}\quad \frac{\Gamma \vdash t_1 : \mathsf{Bool} \qquad \Gamma \vdash t_2 : \mathsf{Bool}}{\Gamma \vdash t_1 == t_2 : \mathsf{Bool}}$$

$$\textsc{True}\quad \frac{}{\Gamma \vdash \textbf{true} : \mathsf{Bool}} \qquad \textsc{False}\quad \frac{}{\Gamma \vdash \textbf{false} : \mathsf{Bool}} \qquad \textsc{And}\quad \frac{\Gamma \vdash t_1 : \mathsf{Bool} \qquad \Gamma \vdash t_2 : \mathsf{Bool}}{\Gamma \vdash t_1\ \&\&\ t_2 : \mathsf{Bool}}$$

$$\textsc{If}\quad \frac{\Gamma \vdash t_1 : \mathsf{Bool} \qquad \Gamma \vdash t_2 : T \qquad \Gamma \vdash t_3 : T}{\Gamma \vdash \textbf{if}\ t_1\ \textbf{then}\ t_2\ \textbf{else}\ t_3 : T} \qquad \textsc{Var}\quad \frac{(x, T) \in \Gamma}{\Gamma \vdash x : T}$$

$$\textsc{Call}\quad \frac{t_1 : T_1 \quad \ldots \quad t_n : T_n \qquad f : (T_1, \ldots, T_n) \Rightarrow T}{f(t_1, \ldots, t_n) : T}$$

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.

Solution

$$\textsc{S-Var1}\ \frac{}{x[x := t_2] \to t_2} \qquad \textsc{S-Var2}\ \frac{x \neq y}{y[x := t_2] \to y} \qquad \textsc{S-Lit}\ \frac{}{n[x := t_2] \to n}$$

$$\textsc{S-True}\ \frac{}{\textbf{true}[x := t_2] \to \textbf{true}} \qquad \textsc{S-False}\ \frac{}{\textbf{false}[x := t_2] \to \textbf{false}}$$

$$\textsc{S-Add}\ \frac{t_1[x := t_3] \to t_1' \qquad t_2[x := t_3] \to t_2'}{(t_1 + t_2)[x := t_3] \to t_1' + t_2'}$$

[similarly for other term shapes]

Question 3

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

Solution

We assume that $\Gamma \vdash t_1 : T$ where $(x : T_x) \in \Gamma$ and $\varepsilon \vdash t_2 : T_x$, meaning that we assume we have derivation trees for these two typing judgments.

We can prove by induction on typing derivations the following lemma: given any Γ', t', and T', if $\varepsilon \vdash t' : T'$ can be derived, then $\Gamma' \vdash t' : T'$ can also be derived (this is a special case of the property known as *context weakening*).

We can get a derivation tree for $\Gamma \vdash t_1[x := t_2] : T_x$ by replacing, in the original derivation tree for $\Gamma \vdash t_1 : T$, all leaves of the form $\Gamma' \vdash x : T_x$ by a derivation tree for $\Gamma' \vdash t_2 : T_x$, which we obtain from the lemma above together with the fact that $\varepsilon \vdash t_2 : T_x$. Note that this can be expressed more formally as another inductive argument on typing derivations.

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.

Solution

As in the lecture, except that the function call reduction rules now look like:

$$\textsc{E-Call}\quad \frac{e(f) = ([x_1,\, \dots,\, x_n],\, [T_1,\, \dots,\, T_n],\, t_{\mathsf{body}},\, T_{\mathsf{result}})}{f(t_1,\, \dots,\, t_n) \rightsquigarrow t_{\mathsf{body}}[x_1 := t_1, \dots, x_n := t_n]}$$

instead of the following (where v denotes *values*):

$$\textsc{E-Call1}\quad \frac{e(f) = ([x_1,\, \dots,\, x_n],\, [T_1,\, \dots,\, T_n],\, t_{\mathsf{body}},\, T_{\mathsf{result}})}{f(v_1,\, \dots,\, v_n) \rightsquigarrow v_{\mathsf{body}}[x_1 := v_1, \dots, x_n := v_n]}$$

$$\textsc{E-Call2}\quad \frac{t_i \rightsquigarrow t_i'}{f(v_1,\, \dots,\, v_{i-1},\, t_i,\, \dots,\, t_n) \rightsquigarrow f(v_1,\, \dots,\, v_{i-1},\, t_i',\, \dots,\, t_n)}$$

Question 5

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

Solution

Progress is straightforward, as the new call-by-name rule always applies. Preservation follows from the fact that substitution preserves types, which was proves in the previous question.