Tutorial 8 (Solutions)

Computer Language Processing and Compiler Design
(COMP 4901U)

November 8, 2021

Exercise (1): Introduction to Subyping

Consider a simple programming language with the following types and terms:

$$\begin{aligned} T &::= \mathsf{Real} \mid \mathsf{Pos} \mid \mathsf{Neg} \\ t &::= c \mid t + t \mid t * t \mid t/t \end{aligned}$$

Real is the type of real numbers, while Pos is the type of strictly positive real numbers and Neg is the type of strictly negative real numbers.

Interestingly, some terms can be assigned multiple types. For instance, 14 has types Real and Pos, while -2 has types Real and Neg. The constant 0, on the other hand, only has type Real.

Question 1

Write down some typing rules for the terms of this language, trying to *preserve information* about positivity and negativity and making sure that your type system *prohibits division by zero.*

Solution

$$\textsc{Real}\ \frac{c \text{ is a number literal}}{c : \mathsf{Real}} \qquad \textsc{Pos}\ \frac{c \text{ is a positive literal}}{c : \mathsf{Pos}} \qquad \textsc{Neg}\ \frac{c \text{ is a negative literal}}{c : \mathsf{Neg}}$$

$$\textsc{Add1}\ \frac{T \in \{\,\mathsf{Pos},\ \mathsf{Neg}\,\} \quad t_1 : T \quad t_2 : T}{t_1 + t_2 : T} \qquad \textsc{Add2}\ \frac{t_1 : \mathsf{Real} \quad t_2 : \mathsf{Real}}{t_1 + t_2 : \mathsf{Real}}$$

$$\textsc{Mul1}\ \frac{T \in \{\,\mathsf{Pos},\ \mathsf{Real}\,\} \quad t_1 : T \quad t_2 : T}{t_1 * t_2 : T} \qquad \textsc{Mul2}\ \frac{t_1 : \mathsf{Pos} \quad t_2 : \mathsf{Neg}}{t_1 * t_2 : \mathsf{Neg}} \qquad \textsc{Mul3}\ \frac{t_1 : \mathsf{Neg} \quad t_2 : \mathsf{Pos}}{t_1 * t_2 : \mathsf{Neg}}$$

$$\textsc{Mul4}\ \frac{t_1 : \mathsf{Neg} \quad t_2 : \mathsf{Neg}}{t_1 * t_2 : \mathsf{Pos}} \qquad \textsc{Div1}\ \frac{t_1 : T \quad t_2 : \mathsf{Pos}}{t_1 / t_2 : T} \qquad \textsc{Div2}\ \frac{t_1 : \mathsf{Pos} \quad t_2 : \mathsf{Neg}}{t_1 / t_2 : \mathsf{Neg}}$$

$$\textsc{Div3}\ \frac{t_1 : \mathsf{Neg} \quad t_2 : \mathsf{Neg}}{t_1 / t_2 : \mathsf{Pos}} \qquad \textsc{Div4}\ \frac{t_1 : \mathsf{Real} \quad t_2 : \mathsf{Neg}}{t_1 / t_2 : \mathsf{Real}}$$

Note that we may be tempted to define rules for all combinations of $\mathsf{Real}+\mathsf{Pos}$, $\mathsf{Real} + \mathsf{Neg}$, $\mathsf{Pos} + \mathsf{Real}$, and $\mathsf{Neg} + \mathsf{Real}$. But in the simplistic type system considered in this exercise, this is in fact unnecessary. Indeed, we can prove (by induction on typing derivations) that for any term t, if $t : \mathsf{Pos}$ or $t : \mathsf{Neg}$ can be derived, then so can $t : \mathsf{Real}$. Now, this property would not hold in a more complicated system where, for example, we would have lambda expressions of the form $(x : T) \Rightarrow t$ along with the usual Var typing rule; in this case, we would need to add the combinatorial rules mentioned above.

Question 2

In your type system, what are the types, if any, of the following terms? Write down a derivation for each possible type.

a. $1 + 1$

b. $-2 * 4$

c. $-1 * (2 + -1)$

d. $7/(18 + -1)$

Solution

(Derivations not shown.)

a. Real, Pos

b. Real, Neg

c. Real

d. No type, since + of positive and negative real numbers is not guaranteed to be non-zero.

Question 3

We now introduce a new relation, written $T <: T$, which we call the *subtyping* relation. The judgment $T_1 <: T_2$ can be read as "T_1 *is a subtype of* T_2". When $T_1 <: T_2$, any terms of type T_1 can safely be used in contexts where terms of type T_2 are expected.

List all pairs of two types of our language which can be made part of this subtyping relation.

Solution

- Real <: Real
- Pos <: Pos
- Neg <: Neg
- Pos <: Real
- Neg <: Real

Question 4

Our goal is now to write a new typing rule, usually called the *subsumption rule*, which bridges the gap between the subtyping and typing relations. This rule should state that if a term has type T_1 and if T_1 is a subtype of T_2, then the term also has type T_2.

a. Write down that rule formally.

b. Now that we have this rule, can some of the previously-defined typing rules be removed as redundant or simplified? Which ones?

Solution

a. $$\textsc{Sub}\quad \frac{t : T_1 \quad T_1 <: T_2}{t : T_2}$$

b. If we had added them before, we can now remove all rules that were special-casing mixing negative and positive numbers. For instance, instead of one rule for adding Real and Pos, one rule for Pos and Real, one rule for Real and Neg, etc., we can simply specify a single rule for adding Real and Real, resulting in type Real, and this holds even in an extended language with typed lambda expressions. (See also the remark in the solution of Question 1.)
Moreover, we can now remove the Real rule, and replace it with the simpler rule below:

$$\textsc{Zero}\quad \frac{}{0 : \mathsf{Real}}$$

Question 5

Let us now expand our language to add a primitive "power" function to it:

$$t \ ::= \ \ldots \mid t \,\hat{}\, t$$

along with the following typing rule:

$$\textsc{Pow}\quad \frac{t_1 : \mathsf{Real} \quad t_2 : \mathsf{Real}}{t_1 \,\hat{}\, t_2 : \mathsf{Real}}$$

Write a typing derivation for the following expression: $(7/2) \,\hat{}\, 3$

Can you think of better typing rules for power expressions?

Solution

The typing derivation is easily derived, for example applying subsumption on all Pos number leaves to make them Real and then using the obvious typing rules.

A more adequate rule for power expressions would be:

$$\frac{t_1 : \mathsf{Pos} \quad t_2 : \mathsf{Real}}{t_1 \,\hat{}\, t_2 : \mathsf{Pos}}$$

where we disallowed the case for non-positive bases, since the operation is not well-defined on real numbers.

Question 6

Are there multiple valid typing derivations that assign the same type to the above expression?

Solution

With the modified typing rules of Question 4, there may be multiples possible derivation trees for a given term and type. Indeed, it is possible to apply the subsumption rule an arbitrary number of times by taking $T_1 = T_2$. Assuming such useless applications of Sub are removed, we do have unique derivations.

Exercise (2): Type Inference

Consider the following type system for a minimal language with anonymous functions and applications:

$$\begin{aligned} t &::= x \mid x \Rightarrow t \mid t\ t \\ T &::= T \to T \mid \alpha \\ \Gamma &::= \varepsilon \mid \Gamma \cdot (x : T) \end{aligned}$$

$$\textsc{Var}\ \frac{(x : T) \in \Gamma}{\Gamma \vdash x : T} \qquad \textsc{Lam}\ \frac{\Gamma \cdot (x : S) \vdash t : T}{\Gamma \vdash x \Rightarrow t : S \to T} \qquad \textsc{App}\ \frac{\Gamma \vdash t_1 : S \to T \quad \Gamma \vdash t_2 : S}{\Gamma \vdash t_1\ t_2 : T}$$

As usual, application has a priority over anonymous function literals, so that, for example, $x \Rightarrow (y \Rightarrow y)\ x$ denotes $x \Rightarrow ((y \Rightarrow y)\ x)$.

Question 1

For each of the following expressions, determine the result of type inference via unification. That is, state whether a most general (i.e., *principal*) type scheme of the form $\forall\overline{\alpha}.\ T$ can be inferred, and if so, write it out. For example, for $(x \Rightarrow x)$ the answer is **yes**, and its most general type scheme is $\forall\alpha.\ \alpha \to \alpha$.

a. $(x \Rightarrow (y \Rightarrow y)\ x)$

b. $(f \Rightarrow x \Rightarrow f\ (f\ x))$

c. $f \Rightarrow x \Rightarrow (g \Rightarrow f\ (g\ x))\ x$

d. $f \Rightarrow g \Rightarrow x \Rightarrow f\ x\ (g\ x)$

e. $f \Rightarrow g \Rightarrow x \Rightarrow g\ (f\ x)$

Solution

a. Yes: $\forall\alpha.\ \alpha \to \alpha$

b. Yes: $\forall\alpha.\ (\alpha \to \alpha) \to \alpha \to \alpha$

c. No: because of the *occurs check.* To intuitively see why this check must fail at some point during unification, note that in $(g \Rightarrow f\ (g\ x))\ x$, we are essentially applying x to itself (indirectly).

d. Yes: $\forall\alpha,\ \beta,\ \gamma.\ (\alpha \to \beta \to \gamma) \to (\alpha \to \beta) \to \alpha \to \gamma$

e. Yes: $\forall\alpha,\ \beta,\ \gamma.\ (\alpha \to \beta) \to (\beta \to \gamma) \to \alpha \to \gamma$

Question 2

To prove that a type scheme $\forall\overline{\alpha}.\ S$ "*is at least as general as*" (i.e., *subsumes*) another type scheme $\forall\overline{\beta}.\ T$, we need to find an instantiation ρ of the $\overline{\alpha}$ variables such that $\rho(S) = T$, where $\rho(S)$ represents the type S after substituting all $\overline{\alpha}$ variables with some other types.[1]

Given:

$$\begin{aligned} \sigma_1 &= \forall\alpha_1,\, \alpha_2.\ (\alpha_1 \to \alpha_2) \to \alpha_1 \to \alpha_2 \\ \sigma_2 &= \forall\beta_1.\ (\beta_1 \to \beta_1) \to \beta_1 \to \beta_1 \end{aligned}$$

a. Prove that σ_1 subsumes σ_2.

b. Prove that that σ_2 *does not* subsume σ_1.

c. Prove that σ_1 is a *principal type scheme* for the term $x \Rightarrow y \Rightarrow x\ y$; that is, prove that any other type scheme that can be assigned to this term is subsumed by σ_1.

Solution

a. Simply take $\alpha_1 = \alpha_2 = \beta_1$.

b. We need to show that there exist *no substitutions* ρ of α_1 and α_2 such that $\rho(S) = T$. This can be shown by contradiction: if there were such a substitution, it would assign β_1 to both α_1 and α_2, which is a contradiction.

c. We need to show that given *any* type scheme $\sigma = \forall\overline{\gamma}.\ T$ that can be assigned to the term at hand, then σ_1 subsumes σ. We can do this by analyzing all the typing rules that can possibly used to derive some type T for the term, and at each step, derive a corresponding substitution from α_1 and α_2 to the corresponding types in the studied typing derivation, so as to make $(\alpha_1 \to \alpha_2) \to \alpha_1 \to \alpha_2$ match it.
Concretely, the last rule used in the derivation must be Lam because the term is a function expression, and this rule must assign some S type to x in the context. So we know α should be mapped to this S.

[1]For simplicity, in this definition we assume that the variables that are quantified in each type do not occur in the other type. Given any two arbitrary types, this can always be ensured through appropriate renaming.

Then, to type the body of the lambda, we can show that this mapping assumption is never contradicted. The same reasoning goes for the rest of the typing derivation and for the result type.

Question 3

Imagine we modify our type system so that polymorphic type schemes are now treated as proper types, which can appear in any position a type T can appear. That is, we extend the language as follows:

$$T \ ::= \ T \to T \mid \alpha \mid \forall \alpha.\ T$$

$$\ldots \qquad \frac{\Gamma \vdash t : T \quad \alpha \text{ does not occur in } \Gamma}{\Gamma \vdash t : \forall \alpha.\ T}\ \textsc{Gen} \qquad \frac{t : \forall \alpha.\ T}{t : T[\alpha \mapsto S]}\ \textsc{Inst}$$

where $T[\alpha \mapsto S]$ denotes the substitution, in T, of all *free* occurrences of α by S. That is, we specifically do not substitute occurrences that are bound in T by a $\forall$. For instance, $((\forall \alpha.\ \alpha) \to \alpha)[\alpha \mapsto S] = (\forall \alpha.\ \alpha) \to S$.

In this system, we can derive that $x \Rightarrow y \Rightarrow x\ y$ has both type $T_1 = (\alpha \to \beta) \to \alpha \to \beta$ and type $T_2 = (\forall \alpha.\ \alpha \to \alpha) \to \beta \to \beta$ — notice that type variables are allowed to be unbound, which is notably the case of β in T_2.

(1) Provide derivations corresponding to the above T_1 and T_2 typings.

(2) Is either T_1 or T_2 a principal type of the term? If not, can you find one?

(3) Provide at least one type derivation for each of the following terms:

a. $x \Rightarrow x\ x$

b. $x \Rightarrow y \Rightarrow z \Rightarrow f \Rightarrow f\ (x\ y)\ (x\ z)$

c. $x \Rightarrow x\ x\ \ldots\ x$ whereby x is applied an arbitrary number of times; try to come up with a finite type of *constant size*, and provide an informal algorithm to construct the corresponding typing derivation.

Solution

(1) The derivations are easy to work back to, given the types.

(2) Neither is principal, since they do not subsume each other. In fact, this term does not have a principal type in this type system. (Note that there are more complex type systems where it does.)

(3) Again, we omit the derivations, as they are easy to derive from the type.

a. $(\forall\alpha.\ \alpha \to \alpha) \to (\forall\alpha.\ \alpha \to \alpha)$

b. $(\alpha \to \beta) \to \alpha \to \alpha \to (\beta \to \beta \to \gamma) \to \gamma$
or $(\forall\alpha.\ \alpha \to \alpha) \to \beta \to \gamma \to (\beta \to \gamma \to \delta) \to \delta$

c. Same idea as before: $(\forall\alpha.\ \alpha \to \alpha) \to (\forall\alpha.\ \alpha \to \alpha)$
or even just $(\forall\alpha.\ \alpha) \to \beta$

Question 4 (hard)

Can you come up with an algorithm to infer a valid typing derivation given a term in our extended language? How about an algorithm to decide whether a term is well-typed, without having to provide any derivations?

Solution

Both questions turn out to be undecidable!
See: undecidability of type inference for System F.