1 Parametric polymorphism

(a) For each of the following System F expressions, is the expression well-typed, and if so, what type does it have? (If you are unsure, try to construct a typing derivation. Make sure you understand the typing rules.)

- $\Lambda A. \lambda x : A \rightarrow \text{int}. \mathbf{42}$
- $\lambda y : \forall X. X \rightarrow X. (y \ [\text{int}]) \ 17$
- $\Lambda Y. \Lambda Z. \lambda f : Y \rightarrow Z. \lambda a : Y. f \ a$
- $\Lambda A. \Lambda B. \Lambda C. \lambda f : A \rightarrow B \rightarrow C. \lambda b : B. \lambda a : A. f \ a \ b$

Answer:

- $\Lambda A. \lambda x : A \rightarrow \text{int}. \mathbf{42}$ has type $\forall A. (A \rightarrow \text{int}) \rightarrow \text{int}$
- $\lambda y : \forall X. X \rightarrow X. (y \ [\text{int}]) \ 17$ has type $(\forall X. X \rightarrow X) \rightarrow \text{int}$
- $\Lambda Y. \Lambda Z. \lambda f : Y \rightarrow Z. \lambda a : Y. f \ a$ has type $\forall Y. \forall Z. (Y \rightarrow Z) \rightarrow Y \rightarrow Z$
- $\Lambda A. \Lambda B. \Lambda C. \lambda f : A \rightarrow B \rightarrow C. \lambda b : B. \lambda a : A. f \ a \ b$ has type $\forall A. \forall B. \forall C. (A \rightarrow B \rightarrow C) \rightarrow B \rightarrow A \rightarrow C$

(b) For each of the following types, write an expression with that type.

- $\forall X. X \rightarrow (X \rightarrow X)$
- $(\forall C. \forall D. C \rightarrow D) \rightarrow (\forall E. \text{int} \rightarrow E)$
- $\forall X. X \rightarrow (\forall Y. Y \rightarrow X)$

Answer:

- $\forall X. X \rightarrow (X \rightarrow X)$ is the type of $\Lambda X. \lambda x : X. \lambda y : X. y$
- $(\forall C. \forall D. C \rightarrow D) \rightarrow (\forall E. \text{int} \rightarrow E)$ is the type of $\lambda f : \forall C. \forall D. C \rightarrow D. \Lambda E. \lambda x : \text{int}. (f \ [\text{int}] \ [E]) \ x$
2 Records and Subtyping

(a) Assume that we have a language with references and records.

(i) Write an expression with type

\( \{ \text{cell} : \text{int ref}, \text{inc} : \text{unit} \rightarrow \text{int} \} \)

such that invoking the function in the field \text{inc} will increment the contents of the reference in the field \text{cell}.

**Answer:** The following expression has the appropriate type.

\[
\text{let } x = \text{ref}\ 14 \text{ in } \\
\{ \text{cell} = x, \text{inc} = \lambda u : \text{unit}. x := (!x + 1) \}
\]

(ii) Assuming that the variable \(y\) is bound to the expression you wrote for part (i) above, write an expression that increments the contents of the cell twice.

**Answer:**

\[
\text{let } z = y.\text{inc} () \text{ in } y.\text{inc} ()
\]

(b) The following expression is well-typed (with type \text{int}). Show its typing derivation. (Note: you will need to use the subsumption rule.)

\[
(\lambda x : \{\text{dogs} : \text{int}, \text{cats} : \text{int}\}. x.\text{dogs} + x.\text{cats}) \{\text{dogs} = 2, \text{cats} = 7, \text{mice} = 19\}
\]

**Answer:**

For brevity, let \(e_1 \equiv \lambda x : \{\text{dogs} : \text{int}, \text{cats} : \text{int}\}. x.\text{dogs} + x.\text{cats}\) and let \(e_2 \equiv \{\text{dogs} = 2, \text{cats} = 7, \text{mice} = 19\}\). The derivation has the following form.

\[
\begin{array}{c}
\vdash e_1 : \{\text{dogs} : \text{int}, \text{cats} : \text{int}\} \rightarrow \text{int} \\
\vdash e_2 : \{\text{dogs} : \text{int}, \text{cats} : \text{int}\}
\end{array}
\]

The derivation of \(e_1\) is straightforward:
The derivation of \( e_2 \) requires the use of subsumption, since we need to show that 
\[ e_2 \equiv \{\text{dogs} = 2, \text{cats} = 7, \text{mice} = 19\} \]
has type \{dogs : int, cats : int\}.

\[
\vdash 2 : \text{int} \\
\vdash 7 : \text{int} \\
\vdash 19 : \text{int}
\]

\[
\vdash \{\text{dogs} = 2, \text{cats} = 7, \text{mice} = 19\} : \{\text{dogs} : \text{int}, \text{cats} : \text{int}, \text{mice} : \text{int}\}
\]

\[
\{\text{dogs} : \text{int}, \text{cats} : \text{int}, \text{mice} : \text{int}\} \leq \{\text{dogs} : \text{int}, \text{cats} : \text{int}\}
\]

\[
\vdash \{\text{dogs} = 2, \text{cats} = 7, \text{mice} = 19\} : \{\text{dogs} : \text{int}, \text{cats} : \text{int}\}
\]

(c) Suppose that \( \Gamma \) is a typing context such that
\[
\Gamma(a) = \{\text{dogs} : \text{int}, \text{cats} : \text{int}, \text{mice} : \text{int}\}
\]
\[
\Gamma(f) = \{\text{dogs} : \text{int}, \text{cats} : \text{int}\} \rightarrow \{\text{apples} : \text{int}, \text{kiwis} : \text{int}\}
\]
Write an expression $e$ that uses variables $a$ and $f$ and has type \{\textit{apples} : \textit{int}\} under context $\Gamma$, i.e., $\Gamma \vdash e : \{\textit{apples} : \textit{int}\}$. Write a typing derivation for it.

**Answer:** A suitable expression is $f \ a$. Note that $f$ is a function that expects an expression of type \{\textit{dogs} : \textit{int}, \textit{cats} : \textit{int}\} as an argument. Variable $a$ is of type \{\textit{dogs} : \textit{int}, \textit{cats} : \textit{int}, \textit{mice} : \textit{int}\}, which is a subtype, so we can use $a$ as an argument to $f$.

Function $f$ returns a value of type \{\textit{apples} : \textit{int}, \textit{kiwis} : \textit{int}\} but our expression $e$ needs to return a value of type \{\textit{apples} : \textit{int}\}. But \{\textit{apples} : \textit{int}, \textit{kiwis} : \textit{int}\} is a subtype of \{\textit{apples} : \textit{int}\}, so it works out.

Here is a typing derivation for it. We abbreviate type \{\textit{dogs} : \textit{int}, \textit{cats} : \textit{int}, \textit{mice} : \textit{int}\} to DCM and abbreviate \{\textit{dogs} : \textit{int}, \textit{cats} : \textit{int}\} to DC.

Which of the inference rules are uses of subsumption? Some of the derivations have been elided. Fill them in.
(d) Which of the following are subtypes of each other?

(a) \{dogs: \texttt{int}, cats: \texttt{int}\} \to \{apples: \texttt{int}\}

(b) \{dogs: \texttt{int}\} \to \{apples: \texttt{int}\}

(c) \{dogs: \texttt{int}\} \to \{apples: \texttt{int}, kiwis: \texttt{int}\}

(d) \{dogs: \texttt{int}, cats: \texttt{int}, mice: \texttt{int}\} \to \{apples: \texttt{int}, kiwis: \texttt{int}\}

(e) \{(apples: \texttt{int})\} \texttt{ref}

(f) \{(apples: \texttt{int}, kiwis: \texttt{int})\} \texttt{ref}

(g) \{(kiwis: \texttt{int}, apples: \texttt{int})\} \texttt{ref}

For each such pair, make sure you have an understanding of why one is a subtype of the other (and for pairs that aren’t subtypes, also make sure you understand).

**Answer:** Of the function types:

- (b) is a subtype of (a)
- (c) is a subtype of (b)
- (c) is a subtype of (d)
- (c) is a subtype of (a)
- (d) is not a subtype of either (a) or (b), or vice versa

The key thing is that for \(\tau_1 \to \tau_2\) to be a subtype of \(\tau'_1 \to \tau'_2\), we must be contravariant in the argument type and covariant in the result type, i.e., \(\tau'_1 \leq \tau_1\) and \(\tau_2 \leq \tau'_2\).

Let’s consider why (b) is a subtype of (a), i.e., \{dogs: \texttt{int}\} \to \{apples: \texttt{int}\} \leq \{dogs: \texttt{int}, cats: \texttt{int}\} \to \{apples: \texttt{int}\}. Suppose we have a function \(f_b\) of type \{dogs: \texttt{int}\} \to \{apples: \texttt{int}\}, and we want to use it somewhere that wants a function \(g_a\) of type \{dogs: \texttt{int}, cats: \texttt{int}\} \to \{apples: \texttt{int}\}. Let’s think about how \(g_a\) could be used: it could be given an argument of type \{dogs: \texttt{int}, cats: \texttt{int}\}, and so \(f_b\) had better be able to handle any record that has the fields dogs and cats. Indeed, \(f_b\) can be given any value of type \{dogs: \texttt{int}\}, i.e., any record that has a field dogs. So \(f_b\) can take any argument that \(g_a\) can be given. The other way that a function can be used is by taking the result of applying it. The result types of the functions are the same, so we have no problem there. Here is a derivation showing the subtyping relation:

\[
\frac{\{dogs: \texttt{int}, cats: \texttt{int}\} \leq \{dogs: \texttt{int}\} \quad \{apples: \texttt{int}\} \leq \{apples: \texttt{int}\}}{\{dogs: \texttt{int}\} \to \{apples: \texttt{int}\} \leq \{dogs: \texttt{int}, cats: \texttt{int}\} \to \{apples: \texttt{int}\}}
\]

Let’s consider why (d) is not a subtype of (a) and (a) is not a subtype of (d). (d) is not a subtype of (a) since they are not contravariant in the argument type (i.e., the argument type of (a) is not a subtype of the argument type of (d)). (a) is not a subtype of (d) since the result type of (a) is not a subtype of the result type of (d) (i.e., they are not covariant in the result type).

For the ref types:

- (f) is a subtype of (g) (and vice versa) assuming the more permissive subtyping rule for records that allows the order of fields to be changed.
- (e) is not a subtype of either (f) or (g), or vice versa.
3 Curry-Howard isomorphism

The following logical formulas are tautologies, i.e., they are true. For each tautology, state the corresponding type, and come up with a term that has the corresponding type.

For example, for the logical formula $\forall \phi. \phi \rightarrow \phi$, the corresponding type is $\forall X. X \rightarrow X$, and a term with that type is $\Lambda X. \lambda x: X. x$. Another example: for the logical formula $\tau_1 \land \tau_2 \Rightarrow \tau_1$, the corresponding type is $\tau_1 \times \tau_2 \rightarrow \tau_1$, and a term with that type is $\lambda x: \tau_1 \times \tau_2. \#1$.

You may assume that the lambda calculus you are using for terms includes integers, functions, products, sums, universal types and existential types.

(a) $\forall \phi. \forall \psi. \phi \land \psi \Rightarrow \psi \lor \phi$

**Answer:** The corresponding type is $\forall X. \forall Y. X \times Y \rightarrow Y + X$

A term with this type is $\Lambda X. \Lambda Y. \lambda x: X \times Y. \text{inl}_X Y \#2 x$

(b) $\forall \phi. \forall \psi. \forall \chi. (\phi \land \psi \Rightarrow \chi) \Rightarrow (\phi \Rightarrow (\psi \Rightarrow \chi))$

**Answer:** The corresponding type is $\forall X. \forall Y. \forall Z. (X \times Y \rightarrow Z) \rightarrow (X \rightarrow (Y \rightarrow Z))$

A term with this type is $\Lambda X. \Lambda Y. \Lambda Z. \lambda f: X \times Y \rightarrow Z. \lambda x: X. \lambda y: Y. f (x, y)$

Note that this term uncurries the function. It is the opposite of the currying we saw in class.

(c) $\exists \phi. \forall \psi. \psi \Rightarrow \phi$

**Answer:** The corresponding type is $\exists X. \forall Y. Y \rightarrow X$

A term with this type is $\text{pack \{ int, } \Lambda Y. \lambda y: Y. 42 \text{ as } \exists X. \forall Y. Y \rightarrow X$

(d) $\forall \psi. \psi \Rightarrow (\forall \phi. \phi \Rightarrow \psi)$

**Answer:** The corresponding type is $\forall Y. Y \rightarrow (\forall X. X \rightarrow Y)$

A term with this type is $\Lambda Y. \lambda a: Y. \Lambda X. \lambda x: X. a$

Primitive propositions in logic correspond
(e) \( \forall \psi. (\forall \phi. \phi \implies \psi) \implies \psi \)

**Answer:** A corresponding type is

\[ \forall Y. (\forall X. X \rightarrow Y) \rightarrow Y \]

A term with this type is

\[ \lambda Y. \lambda f: \forall X. X \rightarrow Y. f \ [\text{int}] \ 42 \]

4 Existential types

(a) Write a term with type \( \exists C. \{ \text{produce} : \text{int} \rightarrow C, \text{consume} : C \rightarrow \text{bool} \} \). Moreover, ensure that calling the function \text{produce} will produce a value of type \( C \) such that passing the value as an argument to \text{consume} will return true if and only if the argument to \text{produce} was 42. (Assume that you have an integer comparison operator in the language.)

**Answer:**

In the following solution, we use \text{int} as the witness type, and implement \text{produce} using the identity function, and implement \text{consume} by testing whether the value of type \( C \) (i.e., of witness type \text{int}) is equal to 42.

\[
\text{pack} \{ \text{int}, \{ \text{produce} = \lambda a: \text{int}. a, \text{consume} = \lambda a: \text{int}. a = 42 \} \}
\]

as \( \exists C. \{ \text{produce} : \text{int} \rightarrow C, \text{consume} : C \rightarrow \text{bool} \} \)

(b) Do the same as in part (a) above, but now use a different witness type.

**Answer:** Here’s another solution where instead we use \text{bool} as the witness type, and implement \text{produce} by comparing the integer argument to 42, and implement \text{consume} as the identity function.

\[
\text{pack} \{ \text{bool}, \{ \text{produce} = \lambda a: \text{int}. a = 42, \text{consume} = \lambda a: \text{bool}. a \} \}
\]

as \( \exists C. \{ \text{produce} : \text{int} \rightarrow C, \text{consume} : C \rightarrow \text{bool} \} \)

(c) Assuming you have a value \( v \) of type \( \exists C. \{ \text{produce} : \text{int} \rightarrow C, \text{consume} : C \rightarrow \text{bool} \} \), use \( v \) to “produce” and “consume” a value (i.e., make sure you know how to use the \text{unpack} \{ X, x \} = e_1 in e_2 expression.

**Answer:**

\[
\text{unpack} \{ D, r \} = v \text{ in }
\]

\[
\text{let} d = r.\text{produce} \ 19 \text{ in }
\]

\[
r.\text{consume} \ d
\]