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.)

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

**Answer:**

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

(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:**

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

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

(i) Write an expression with type

{ cell : int ref, inc : unit → int }

such that invoking the function in the field inc will increment the contents of the reference in the field cell.

Answer: The following expression has the appropriate type.

let x = ref 14 in
{ cell = x, inc = λu: 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: let z = y.inc () in y.inc ()

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

(λx : {dogs : int, cats : int}. x.dogs + x.cats) {dogs = 2, cats = 7, mice = 19}

Answer:

For brevity, let e_1 ≡ λx : {dogs : int, cats : int}. x.dogs + x.cats and let e_2 ≡ {dogs = 2, cats = 7, mice = 19}. The derivation has the following form.

\[ \vdash e_1 : \text{int} \]

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 $\{\text{dogs} : \text{int}, \text{cats} : \text{int}\}$.

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

(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 $\{\text{apples : int}\}$ under context $\Gamma$, i.e., $\Gamma \vdash e : \{\text{apples : int}\}$. 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 $\{\text{dogs : int, cats : int}\}$ as an argument. Variable $a$ is of type $\{\text{dogs : int, cats : int, mice : int}\}$, which is a subtype, so we can use $a$ as an argument to $f$.

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

Here is a typing derivation for it. We abbreviate type $\{\text{dogs : int, cats : int, mice : int}\}$ to DCM and abbreviate $\{\text{dogs : int, cats : int}\}$ 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) \{\text{dogs}: \text{int}, \text{cats}: \text{int}\} \rightarrow \{\text{apples}: \text{int}\}
(b) \{\text{dogs}: \text{int}\} \rightarrow \{\text{apples}: \text{int}\}
(c) \{\text{dogs}: \text{int}\} \rightarrow \{\text{apples}: \text{int}, \text{kiwis}: \text{int}\}
(d) \{\text{dogs}: \text{int}, \text{cats}: \text{int}, \text{mice}: \text{int}\} \rightarrow \{\text{apples}: \text{int}, \text{kiwis}: \text{int}\}
(e) (\{\text{apples}: \text{int}\}) \text{ref}
(f) (\{\text{apples}: \text{int}, \text{kiwis}: \text{int}\}) \text{ref}
(g) (\{\text{kiwis}: \text{int}, \text{apples}: \text{int}\}) \text{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 \(\tau_1 \rightarrow \tau_2\) to be a subtype of \(\tau'_1 \rightarrow \tau'_2\), 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., \(\{\text{dogs}: \text{int}\} \rightarrow \{\text{apples}: \text{int}\} \leq \{\text{dogs}: \text{int}, \text{cats}: \text{int}\} \rightarrow \{\text{apples}: \text{int}\}\). Suppose we have a function \(f_h\) of type \(\{\text{dogs}: \text{int}\} \rightarrow \{\text{apples}: \text{int}\}\), and we want to use it somewhere that wants a function \(g_a\) of type \(\{\text{dogs}: \text{int}, \text{cats}: \text{int}\} \rightarrow \{\text{apples}: \text{int}\}\). Let’s think about how \(g_a\) could be used: it could be given an argument of type \(\{\text{dogs}: \text{int}, \text{cats}: \text{int}\}\), and so \(f_h\) had better be able to handle any record that has the fields dogs and cats. Indeed, \(f_h\) can be given any value of type \(\{\text{dogs}: \text{int}\}\), i.e., any record that has a field dogs. So \(f_h\) 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:

\[
\begin{align*}
\{\text{dogs}: \text{int}, \text{cats}: \text{int}\} & \leq \{\text{dogs}: \text{int}\} \\
\{\text{apples}: \text{int}\} & \leq \{\text{apples}: \text{int}\}
\end{align*}
\]

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

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 $\forall \phi. \phi \implies \phi$, the corresponding type is $\forall X. X \to X$, and a term with that type is $\lambda x : X. x$. Another example: for the logical formula $\tau_1 \land \tau_2 \implies \tau_1$, the corresponding type is $\tau_1 \times \tau_2 \to \tau_1$, and a term with that type is $\lambda x : \tau_1 \times \tau_2. x$.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

A term with this type is

\[ \Lambda Y. \lambda f : \forall X. X \to Y. f [\text{int}] ~ 42 \]

### 4 Existential types

(a) Write a term with type \( \exists C. \{ \text{produce} : \text{int} \to C, \text{consume} : C \to \text{bool} \} \). 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 \( \exists C. \{ \text{produce} : \text{int} \to C, \text{consume} : C \to \text{bool} \} \)

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

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

**Answer:**

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

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

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