CS 599 A1: Assignment 5

Due Friday, April 17, 2025

Total: 100 pts

Ankush Das

This assignment is just programming! You will be implementing a session type checker and interpreter that will follow the typing and semantics rules respectively. Please follow the guidelines below to make sure your submission can be accepted by the instructor.

- The only languages you can use for this assignment are C++, Python, and OCaml.
- In this set of problems, you will be required to define several types and functions.
- Please make sure your submission is a zip file that contains the code file(s) that defines the required functions and types. You should submit the zip file containing the file(s) electronically on Gradescope by the due date.
- Your zip file must include a separate readme file that contains instructions for installing and executing the file(s) in your submission.
- You're welcome to modularize your code into multiple files. If you do so, please indicate in your instructions which file contains the functions and types for each problem.
- For all the following problems, you are welcome to define as many helper functions as needed. Please indicate in a comment what the function's purpose is.
- Begin coding!

1 Type Equality [20 pts]

Before we implement the type checker and interpreter, we will implement the type equality algorithm discussed in class. This algorithm will be implemented as a function that would take two types and return a boolean depending upon whether they are equal or not.

Recall the type syntax of the language.

$$\text{Types} \quad A, B ::= \oplus\{\ell : A_\ell\}_{\ell \in L} \mid \&\{\ell : A_\ell\}_{\ell \in L} \mid A \otimes B \mid A \multimap B \mid \mathbf{1} \mid V$$

Note that there is an additional type in the type grammar denoted by V. This denotes type names like `nat` and `bin`.

Problem 1 (3 pts) *Define a type called* `tp` *for types in the language. Also, define a type called* `tp_def` *to represent a type definition in the program. In general, type definitions have the form:*

```
type x = A
```

For instance, for the following program:

```
type nat = +{succ : nat, zero : 1}
type bin = +{b0 : bin, b1 : bin, e : 1}
```

there are two type definitions: type `nat` *maps to* `+{succ : nat, zero : 1}` *and type* `bin` *maps to* `+{b0 : bin, b1 : bin, e : 1}`*. The set of all type definitions are collected in a type called* `environment`*. This would be helpful in implementing the type equality algorithm.*

Problem 2 (15 pts) *Define a function called* `eq_tp` *with the following signature:*

```
eq_tp: environment -> constraints -> tp -> tp -> bool
```

Here, `environment` *should contain the set of all type definitions in the program. And* `constraints` *stores the equality constraints encountered so far (equivalent to Θ in the class). Choose appropriate data structures for the types* `environment` *and* `constraints`*.*

The function takes the environment, the constraints encountered, and two types as input and returns `true` *if the types are equal and* `false` *otherwise.*

Problem 3 (2 pts) *Test out the* `eq_tp` *function defined above on the following types.*

```
type nat = +{succ : nat, zero : 1}
type nat1 = +{succ : nat1, zero : 1}
type nat2 = +{succ : nat3, zero : 1}
type nat3 = +{succ : nat2, zero : 1}
type even = +{succ : odd, zero : 1}
type odd = +{succ : even}
type even1 = +{succ : +{succ : even1}, zero : 1}
type odd1 = +{succ : +{succ : odd1, zero : 1}}
```

Which two of these types are equal? Use an empty set of constraints to call the `eq_tp` *function since we start without knowing which two of these types are equal.*

2 Type Checker [40 pts]

Equipped with a type equality algorithm, we will now implement the type checker. First, we define a type for process expressions. Recall the process expression syntax of the language:

$$\begin{array}{lll} \text{Expressions} & P ::= & x.\mathsf{k}\,;\,P \mid \mathsf{case}\ x\ (\ell \Rightarrow P_\ell)_{\ell \in L} \mid y \leftarrow \mathsf{recv}\ x\,;\,P \mid \mathsf{send}\ x\ y\,;\,P \mid \mathsf{wait}\ x\,;\,P \mid \mathsf{close}\ x \\ & \mid & x \leftrightarrow y \mid x \leftarrow f\ \overline{y}\,;\,P \end{array}$$

Problem 4 (3 pts) *Define a type called* `exp` *for expressions in the language.*

Problem 5 (2 pts) *Extend the type* `environment` *to contain process declarations and definitions. Remember that process declarations have the form*

```
decl f : (x1 : A1), (x2 : A2), ... (xn : An) |- (x : A)
```

and process definitions have the form

```
proc x <- f x1 x2 ... xn = P
```

Now, we have completed the setup for implementing the type checker. Recall the typing judgment is expressed as follows: $\Sigma\,;\,\Delta \vdash P :: (z : C)$ where

- Σ is the environment containing all the type definitions, and process definitions and declarations.
- Δ is the typing context mapping channel names to session types.
- P is the process expression.
- z is the offered channel name.
- C is the offered channel type.

Problem 6 (35 pts) *Implement a function called* `typecheck` *with the following signature:*

```
typecheck: environment -> context -> exp -> channel -> tp -> bool
```

Choose appropriate types for `context` *and* `channel` *that represent Δ and z respectively in the judgment. As is standard, the function returns* `true` *if the process is well-typed and* `false` *otherwise.*

3 Interpreter [30 pts]

Finally, we will implement an interpreter that will follow the rules of the semantics. To express the semantics, we first need to define semantic objects: $\mathsf{proc}(c, P)$ and $\mathsf{msg}(c, M)$. Recall the grammar for processes and messages.

$$
\begin{array}{rl}
\text{Expressions} & P ::= x.\mathsf{k}\,;\,P \mid \mathsf{case}\ x\ (\ell \Rightarrow P_\ell)_{\ell \in L} \mid y \leftarrow \mathsf{recv}\ x\,;\,P \mid \mathsf{send}\ x\ y\,;\,P \mid \mathsf{wait}\ x\,;\,P \mid \mathsf{close}\ x \\
& \quad \mid\ \ x \leftrightarrow y \mid x \leftarrow f\ \overline{y}\,;\,P \\
\text{Messages} & M ::= x.\mathsf{k}\,;\,x \leftrightarrow x' \mid x.\mathsf{k}\,;\,x' \leftrightarrow x \mid \mathsf{send}\ x\ y\,;\,x \leftrightarrow x' \mid \mathsf{send}\ x\ y\,;\,x' \leftrightarrow x \mid \mathsf{close}\ x
\end{array}
$$

Problem 7 (5 pts) *Define a type `sem` that represents a semantic object: either a process or a message. Also, define a type `configuration` that represents a set of semantic objects.*

Problem 8 (15 pts) *Implement a function called `step` with the following signature:*

```
step: environment -> configuration -> configuration
```

Problem 9 (10 pts) *Demonstrate the progress theorem by defining the following functions:*

```
poised_sem: sem -> bool
poised: configuration -> bool
```

The first function takes a semantic object (i.e., a process or a message) as input and returns whether it is poised or not. The second function applies the former pointwise to every object in the configuration because a configuration is poised when all its semantic objects are poised.

Recall the definition of poised: a process is poised if it is receiving on the channel it is offering. A message is poised if it is sending on the channel it is offering.

4 Testing [10 pts]

With all components implemented, we will now test out our language functions. For this, we need a way of executing a closed process. We will introduce a new declaration and add it to the environment. This is written as

```
exec f
```

which stands for executing process f.

Problem 10 (1 pts) *Extend the type `environment` to account for process execution declarations. This declaration just contains a process name.*

Problem 11 (9 pts) *Consider the following program:*

```
type nat = +{succ : nat, zero : 1}

decl two : . |- (x : nat)
proc x <- two = x.succ; x.succ; close x

decl plus : (x : nat) (y : nat) |- (z : nat)
proc z <- plus x y =
  case x (
    succ => z.succ ; z' <- plus x y ; z <-> z'
  | zero => wait x ; z <-> y
  )

decl ten : . |- (x : nat)
proc x <- ten =
  x2 <- two; y2 <- two;
  x4 <- plus x2 y2;
  x2 <- two; y2 <- two; z2 <- two;
```

```
y4 <- plus x2 y2;
x6 <- plus y4 z2;
x' <- plus x4 x6;
x <-> x'
```

```
exec ten
```

Write this program (by creating a value of type `environment`*) in your language and print the output of executing the process* `ten`*. For this problem, you should define the following process*

```
string_of_configuration: configuration -> string
```

and print out each intermediate configuration. Confirm that the final configuration is poised.

A Type System, Type Equality, and Semantics

Type System

$$\frac{(k \in L) \qquad \Delta \vdash P :: (x : A_k)}{\Delta \vdash (x.k\,;\, P) :: (x : \oplus\{\ell : A_\ell\}_{\ell \in L})}\ \oplus R \qquad \frac{(\forall \ell \in L) \qquad \Delta, x : A_\ell \vdash Q_\ell :: (z : C)}{\Delta, x : \oplus\{\ell : A_\ell\}_{\ell \in L} \vdash (\mathsf{case}\ x\ (\ell \Rightarrow Q_\ell)_{\ell \in L}) :: (z : C)}\ \oplus L$$

$$\frac{(\forall \ell \in L) \qquad \Delta \vdash P_\ell :: (x : A_\ell)}{\Delta \vdash (\mathsf{case}\ x\ (\ell \Rightarrow P_\ell)_{\ell \in L)} :: (x : \&\{\ell : A_\ell\}_{\ell \in L})}\ \& R \qquad \frac{(k \in L) \qquad \Delta, x : A_k \vdash Q :: (z : C)}{\Delta, x : \&\{\ell : A_\ell\}_{\ell \in L} \vdash (x.k\,;\, Q) :: (z : C)}\ \& L$$

$$\frac{\Delta \vdash P :: (x : B)}{\Delta, y : A \vdash (\mathsf{send}\ x\ y\,;\, P) :: (x : A \otimes B)}\ \otimes R \qquad \frac{\Delta, y : A, x : B \vdash Q :: (z : C)}{\Delta, x : A \otimes B \vdash (y \leftarrow \mathsf{recv}\ x\,;\, Q) :: (z : C)}\ \otimes L$$

$$\frac{\Delta, y : A \vdash P :: (x : B)}{\Delta \vdash (y \leftarrow \mathsf{recv}\ x\,;\, P) :: (x : A \multimap B)}\ \multimap R \qquad \frac{\Delta, x : B \vdash Q :: (z : C)}{\Delta, x : A \multimap B, y : A \vdash (\mathsf{send}\ x\ y\,;\, Q) :: (z : C)}\ \multimap L$$

$$\frac{}{\cdot \vdash (\mathsf{close}\ x) :: (x : \mathbf{1})}\ \mathbf{1}R \qquad \frac{\Delta \vdash Q :: (z : C)}{\Delta, x : \mathbf{1} \vdash (\mathsf{wait}\ x\,;\, Q) :: (z : C)}\ \mathbf{1}L \qquad \frac{}{x : A \vdash (y \leftrightarrow x) :: (y : A)}\ \mathsf{id}$$

$$\frac{\mathsf{decl}\ f : \overline{y' : A'} \vdash (x : A) \in \Sigma \qquad \Delta, x : A \vdash Q :: (z : C)}{\Delta, \overline{y : A'} \vdash (x \leftarrow f\ \overline{y}\,;\, Q) :: (z : C)}\ \mathsf{def}$$

Type Equality

$$\frac{L = M \qquad (\forall \ell \in L)\ \Theta \vdash A_\ell \equiv B_\ell}{\Theta \vdash \oplus\{\ell : A_\ell\}_{\ell \in L} \equiv \oplus\{m : B_m\}_{m \in M}}\ \oplus \qquad \frac{L = M \qquad (\forall \ell \in L)\ \Theta \vdash A_\ell \equiv B_\ell}{\Theta \vdash \&\{\ell : A_\ell\}_{\ell \in L} \equiv \&\{m : B_m\}_{m \in M}}\ \&$$

$$\frac{\Theta \vdash A_1 \equiv B_1 \qquad \Theta \vdash A_2 \equiv B_2}{\Theta \vdash A_1 \otimes A_2 \equiv B_1 \otimes B_2}\ \otimes \qquad \frac{\Theta \vdash A_1 \equiv B_1 \qquad \Theta \vdash A_2 \equiv B_2}{\Theta \vdash A_1 \multimap A_2 \equiv B_1 \multimap B_2}\ \multimap \qquad \frac{}{\Theta \vdash \mathbf{1} \equiv \mathbf{1}}\ \mathbf{1}$$

$$\frac{(V \equiv B) \notin \Theta \qquad \mathsf{type}\ V = A \in \Sigma \qquad \Theta, (V \equiv B) \vdash A \equiv B}{\Theta \vdash V \equiv B}\ \mathsf{expd}R$$

$$\frac{(V \equiv B) \notin \Theta \qquad \mathsf{type}\ V = A \in \Sigma \qquad \Theta, (V \equiv B) \vdash B \equiv A}{\Theta \vdash B \equiv V}\ \mathsf{expd}L$$

$$\frac{(V \equiv B) \in \Theta}{\Theta \vdash V \equiv B}\ \mathsf{def}R \qquad \frac{(V \equiv B) \in \Theta}{\Theta \vdash B \equiv V}\ \mathsf{def}L$$

Two types A and B are said to be equal iff we can derive $\cdot \vdash A \equiv B$.

Semantics

$$
\begin{array}{lll}
(\oplus S) & \mathsf{proc}(c, c.k\,;\, P) \mapsto \mathsf{proc}(c', [c'/c]P), \mathsf{msg}(c, c.k\,;\, c \leftrightarrow c') & (c' \text{ fresh}) \\
(\oplus C) & \mathsf{msg}(c, c.k\,;\, c \leftrightarrow c'), \mathsf{proc}(d, \mathsf{case}\ c\ (\ell \Rightarrow Q_\ell)_{\ell \in L}) \mapsto \mathsf{proc}(d, [c'/c]Q_k) & \\
(\,\&\, S) & \mathsf{proc}(d, c.k\,;\, Q) \mapsto \mathsf{msg}(c', c.k\,;\, c' \leftrightarrow c), \mathsf{proc}(d, [c'/c]Q) & (c' \text{ fresh}) \\
(\,\&\, C) & \mathsf{proc}(c, \mathsf{case}\ c\ (\ell \Rightarrow Q_\ell)_{\ell \in L}), \mathsf{msg}(c', c.k\,;\, c' \leftrightarrow c) \mapsto \mathsf{proc}(c', [c'/c]Q_k) & \\
(\otimes S) & \mathsf{proc}(c, \mathsf{send}\ c\ e\,;\, P) \mapsto \mathsf{proc}(c', [c'/c]P), \mathsf{msg}(c, \mathsf{send}\ c\ e\,;\, c \leftrightarrow c') & (c' \text{ fresh}) \\
(\otimes C) & \mathsf{msg}(c, \mathsf{send}\ c\ e\,;\, c \leftrightarrow c'), \mathsf{proc}(d, x \leftarrow \mathsf{recv}\ c\,;\, Q) \mapsto \mathsf{proc}(d, [c', e/c, x]Q) & \\
(\multimap S) & \mathsf{proc}(d, \mathsf{send}\ c\ e\,;\, Q) \mapsto \mathsf{msg}(c', \mathsf{send}\ c\ e\,;\, c' \leftrightarrow c), \mathsf{proc}(d, [c'/c]Q) & (c' \text{ fresh}) \\
(\multimap C) & \mathsf{proc}(c, x \leftarrow \mathsf{recv}\ c), \mathsf{msg}(c', \mathsf{send}\ c\ e\,;\, c' \leftrightarrow c) \mapsto \mathsf{proc}(c', [c', d/c, x]P) & \\
(\mathbf{1}S) & \mathsf{proc}(c, \mathsf{close}\ c) \mapsto \mathsf{msg}(c, \mathsf{close}\ c) & \\
(\mathbf{1}C) & \mathsf{msg}(c, \mathsf{close}\ c), \mathsf{proc}(d, \mathsf{wait}\ c\,;\, Q) \mapsto \mathsf{proc}(d, Q) & \\
(\mathsf{def}C) & \mathsf{proc}(d, x \leftarrow P_x\ \overline{y}\,;\, Q_x) \mapsto \mathsf{proc}(c, [c/x]P_x), \mathsf{proc}(d, [c/x]Q_x) & (c \text{ fresh}) \\
(\mathsf{id}^+ C) & \mathsf{msg}(d, M), \mathsf{proc}(c, c \leftrightarrow d) \mapsto \mathsf{msg}(c, [c/d]M) & \\
(\mathsf{id}^- C) & \mathsf{proc}(c, c \leftrightarrow d), \mathsf{msg}(e, M_c) \mapsto \mathsf{msg}(e, [d/c]M_c) &
\end{array}
$$