CSED321 Assignment 9 - *Polymorphism*

Due Saturday, June 1

Welcome to the final assignment of CSED321! In this assignment, you will implement a type reconstruction algorithm for Tiny ML (*TML*) from the previous assignment.

1 Syntax

The concrete grammar of TML, introduced in the previous assignment, is defined as follows:

$$
\begin{array}{rl}
\mathit{sconty} ::= & \textbf{int} \mid \textbf{bool} \mid \textbf{unit} \\
\mathit{sconpat} ::= & \mathit{num} \mid \textbf{true} \mid \textbf{false} \mid () \\
\mathit{scon} ::= & \mathit{num} \mid \textbf{true} \mid \textbf{false} \mid () \\
\mathit{op} ::= & + \mid - \mid * \mid = \mid < \\
\mathit{ty} ::= & \mathit{sconty} \mid \mathit{tycon} \mid \mathit{ty} * \mathit{ty} \mid \mathit{ty} \texttt{ -> } \mathit{ty} \mid \mathit{ty}\ \textbf{ref} \mid (\mathit{ty}) \\
\mathit{pat} ::= & \_ \mid \mathit{sconpat} \mid \mathit{vid} \mid \mathit{cid}\ \langle \mathit{pat} \rangle \mid (\mathit{pat}, \mathit{pat}) \mid (\mathit{pat}) \mid (\mathit{pat} : \mathit{ty}) \\
\mathit{mrule} ::= & \mathit{pat} \texttt{ -> } \mathit{exp} \\
\mathit{mlist} ::= & \mathit{mrule}\ \langle \texttt{|}\ \mathit{mlist} \rangle \\
\mathit{exp} ::= & \mathit{scon} \mid \mathit{vid} \mid \mathit{cid} \mid \mathit{exp}\ \mathit{exp} \mid \mathit{exp}\ \mathit{op}\ \mathit{exp} \mid (\mathit{exp}, \mathit{exp}) \\
\mid & \textbf{ref}\ \mathit{exp} \mid !\,\mathit{exp} \mid \mathit{exp} := \mathit{exp} \mid \textbf{fun}\ \mathit{mrule} \mid \textbf{match}\ \mathit{exp}\ \textbf{with}\ \mathit{mlist} \\
\mid & \textbf{let}\ \mathit{pat} = \mathit{exp}\ \textbf{in}\ \mathit{exp} \mid \textbf{let rec}\ \mathit{pat} = \mathit{exp}\ \textbf{in}\ \mathit{exp} \mid (\mathit{exp}) \mid (\mathit{exp} : \mathit{ty}) \\
\mathit{conbinding} ::= & \mathit{cid}\ \langle \textbf{of}\ \mathit{ty} \rangle \\
\mathit{conbind} ::= & \mathit{conbinding}\ \langle \texttt{|}\ \mathit{conbind} \rangle \\
\mathit{dec} ::= & \textbf{type}\ \mathit{tycon} = \mathit{conbind} \\
\mathit{dlist} ::= & \langle \mathit{dec} \rangle * \\
\mathit{program} ::= & \langle \mathit{dlist}\ ;; \rangle\ \mathit{exp}
\end{array}
$$

Each grammar rule consists of a grammar variable on the left side and its expansion forms on the right side. Expansion forms are separated by |. For example, *scon* can be expanded into *num*, **true**, **false** or (). A pair of brackets ⟨⟩ encloses an optional phrase. E.g., *conbinding* is expanded into either *cid* or *cid* followed by **of** *ty*. Starred brackets ⟨⟩∗ represent zero or more repetitions of the enclosed phrase. E.g., *dlist* can have zero or more *dec*'s. A declaration list and an expression in a top-level program is separated by ;; (for parsing in OCaml).

The file `tml.ml` defines types for the grammar variables in the structure `Tml`. The file `validate.ml` defines functions for syntactic validation. Specifically, the function `vprogram` takes a `Tml.program` value, checks whether the `Tml.program` value obeys the syntactic restrictions, and returns the `Tml.program` value. It raises `AstValidateError` if the `Tml.program` value violates the syntactic restrictions. Finally, the file `print.ml` defines functions for printing the content of grammar variables in the structure `Print`. The meaning of each function in the structure should be clear; for example, `exp2str exp` returns a string representation of the `Tml.exp` value `exp`. These functions can be useful when debugging your code.

2 Typing

The goal of this assignment is to implement the function `tprogram` in the file `hw9.ml` for a type reconstruction algorithm of TML. We will not guide you through the whole implementation; we will provide you with the static semantics of TML and some hints about how to design an algorithm. You will have to design and implement your own type reconstruction algorithm.

First read the description of the static semantics, and think about how to design a type reconstruction algorithm. Do *not* jump to coding right after reading the description of the static semantics. We suggest that you spend enough time understanding the static semantics and designing a type reconstruction algorithm before you start any coding.

2.1 Semantic Objects

We refer to all the grammar variables in Section 1 as *syntactic objects*. In addition, there are several *semantic objects* that state the meaning of these syntactic objects.

Semantic Type. Semantic type τ is defined as follows, in contrast to *syntactic type ty*:

$$
\begin{array}{rcll}
\tau & ::= & \mathsf{int} & \text{integer type} \\
 & | & \mathsf{bool} & \text{boolean type} \\
 & | & \mathsf{unit} & \text{unit type} \\
 & | & t & \text{type name} \\
 & | & \tau \times \tau & \text{pair type} \\
 & | & \tau \rightarrow \tau & \text{function type} \\
 & | & \mathsf{ref}\ \tau & \text{reference type} \\
 & | & \alpha & \text{type variable}
\end{array}
$$

Semantic type τ differs from syntactic type *ty*. A syntactic type denotes string expressions for types in TML programs, while a semantic type denotes meaningful types in TML programs. For example, consider the following TML program:

```
type t = T1
type t = T2 ;;
match 1 with 0 -> T1 | _ -> T2
```

The expression **`match`** `1` **`with`** `0 -> T1 | _ -> T2` should typecheck syntactically, since both `T1` and `T2` have the same type constructor `t`. This program, however, does *not* typecheck semantically, since the static semantics of TML (see the rule (40) in Section 2.2) gives different type names for the first type constructor `t` and the second type constructor `t`.

Type Scheme. Type scheme σ is defined as $\forall\vec{\alpha}.\,\tau$, where $\vec{\alpha}$ denotes a set of type variables. We simply write $\forall.\,\tau$ for a type scheme $\forall\vec{\alpha}.\,\tau$ with $\vec{\alpha} = \emptyset$, and write $\forall\alpha.\,\tau$ for a type scheme $\forall\vec{\alpha}.\,\tau$ with $\vec{\alpha} = \{\alpha\}$ for some type variable α.

Note that $\forall\alpha.\,\alpha \rightarrow \alpha$ can be instantiated into any of $\mathsf{int} \rightarrow \mathsf{int}$ and $\mathsf{bool} \rightarrow \mathsf{bool}$ simultaneously, whereas $\forall.\,\alpha \rightarrow \alpha$ can be instantiated into only one of them. For example, consider:

```
(f 1, f true)
```

If `f` has type scheme $\forall\alpha.\,\alpha \rightarrow \alpha$, `f` has type $\mathsf{int} \rightarrow \mathsf{int}$ at `f 1` and $\mathsf{bool} \rightarrow \mathsf{bool}$ at `f true`. On the other hands, if `f` has type scheme $\forall.\,\alpha \rightarrow \alpha$, the type of `f` permanently becomes $\mathsf{int} \rightarrow \mathsf{int}$ at `f 1`, so `f true` does not typecheck since `f` has type $\mathsf{int} \rightarrow \mathsf{int}$ and `true` has type bool.

Typing Context. A typing context Γ is a set of type bindings and type declaration bindings. A type binding $id : \sigma$ means that (variable or constructor) identifier id has type scheme σ. A type declaration binding $tycon \mapsto t$ means that a type constructor $tycon$ is related to type name t. Both bindings connect *syntactic* object with *semantic* object.

$$\text{typing context} \qquad \Gamma \ ::= \ \cdot \ | \ \Gamma, id : \sigma \ | \ \Gamma, tycon \mapsto t$$

As usual, we assume that identifiers and type constructors in Γ are all distinct; for example, $\Gamma, id : \sigma$ is not defined if Γ contains another $id : \sigma'$. We write $\Gamma \oplus \Gamma'$ for the union of Γ and Γ', where for *duplicate* type (declaration) bindings that appear in both Γ and Γ', only those in Γ' are included in $\Gamma \oplus \Gamma'$. For example, $(x : \sigma_1) \oplus (x : \sigma_2) = x : \sigma_2$.

2.2 Typing Rules

Typing Judgement. The static semantics uses the following form of typing judgments.

- $\Gamma \vdash ty \succ \tau$ means that type ty has type τ under typing context Γ.
- $\Gamma \vdash pat \succ \tau, \Gamma'$ means that pattern pat has type τ and gives type bindings in Γ' under Γ.
- $\Gamma \vdash exp \succ \tau$ means that expression exp has type τ under typing context Γ.
- $\Gamma \vdash mlist \succ \tau$ means that matching rule list $mlist$ has type τ under typing context Γ.
- $\Gamma \vdash dlist \succ \Gamma'$ means that declaration list $dlist$ introduces type bindings and type declaration bindings in Γ' under typing context Γ.
- $\Gamma \vdash_t conbind \succ \Gamma'$ means that constructor binding list $conbind$ introduces type bindings in Γ' with associated type name t under typing context Γ.

Rules for Syntactic Types. The following inference rules are for $\Gamma \vdash ty \succ \tau$.

$$\frac{}{\Gamma \vdash \textbf{int} \succ \mathsf{int}}\ (1) \qquad \frac{}{\Gamma \vdash \textbf{bool} \succ \mathsf{bool}}\ (2) \qquad \frac{}{\Gamma \vdash \textbf{unit} \succ \mathsf{unit}}\ (3) \qquad \frac{tycon \mapsto t \in \Gamma}{\Gamma \vdash tycon \succ t}\ (4)$$

$$\frac{\Gamma \vdash ty_1 \succ \tau_1 \quad \Gamma \vdash ty_2 \succ \tau_2}{\Gamma \vdash (ty_1 * ty_2) \succ \tau_1 \times \tau_2}\ (5) \qquad \frac{\Gamma \vdash ty_1 \succ \tau_1 \quad \Gamma \vdash ty_2 \succ \tau_2}{\Gamma \vdash (ty_1 \texttt{ -> } ty_2) \succ \tau_1 \rightarrow \tau_2}\ (6) \qquad \frac{\Gamma \vdash ty \succ \tau}{\Gamma \vdash ty \ \textbf{ref} \succ \mathsf{ref}\ \tau}\ (7)$$

- (1)–(3): each special constant type has the predefined semantic type.
- (4): $tycon$ has type t if $tycon$ with type name t appears in Γ.
- (5): if ty_i has type τ_i for $i = 1, 2$, then $ty_1 * ty_2$ has type $\tau_1 \times \tau_2$.
- (6): if ty_i has type τ_i for $i = 1, 2$, then $ty_1 \texttt{ -> } ty_2$ has type $\tau_1 \rightarrow \tau_2$.
- (7): if ty has type τ, then ty **ref** has type $\mathsf{ref}\ \tau$.

Rules for Patterns. The following inference rules are for $\Gamma \vdash pat \succ \tau, \Gamma'$, where α fresh means that α is a fresh type variable which does not appear in Γ:

$$\frac{\alpha\ \mathsf{fresh}}{\Gamma \vdash \_ \succ \alpha, \cdot}\ (8) \qquad \frac{}{\Gamma \vdash num \succ \mathsf{int}, \cdot}\ (9) \qquad \frac{}{\Gamma \vdash () \succ \mathsf{unit}, \cdot}\ (10)$$

$$\frac{}{\Gamma \vdash \textbf{true} \succ \mathsf{bool}, \cdot}\ (11) \qquad \frac{}{\Gamma \vdash \textbf{false} \succ \mathsf{bool}, \cdot}\ (12)$$

$$\frac{\alpha\ \mathsf{fresh}}{\Gamma \vdash vid \succ \alpha, vid : \forall. \alpha}\ (13) \qquad \frac{cid : \forall. t \in \Gamma}{\Gamma \vdash cid \succ t, \cdot}\ (14) \qquad \frac{cid : \forall. \tau \rightarrow t \in \Gamma \quad \Gamma \vdash pat \succ \tau, \Gamma'}{\Gamma \vdash cid\ pat \succ t, \Gamma'}\ (15)$$

$$\frac{\Gamma \vdash pat_1 \succ \tau_1, \Gamma_1 \quad \mathrm{C} \vdash pat_2 \succ \tau_2, \Gamma_2}{\Gamma \vdash (pat_1, pat_2) \succ \tau_1 \times \tau_2, \Gamma_1 \oplus \Gamma_2}\ (16) \qquad \frac{\Gamma \vdash pat \succ \tau, \Gamma' \quad \Gamma \vdash ty \succ \tau}{\Gamma \vdash (pat : ty) \succ \tau, \Gamma'}\ (17)$$

- (8): _ has fresh type α (i.e., the type is not known yet), and introduces no type binding.
- (9)–(12): each special constant pattern has the predefined type and gives no type binding.
- (13): *vid* has fresh type α, and introduces new type binding $vid : \forall. \alpha$.
- (14): if *cid* has type scheme $\forall.t$ for type name t, then *cid* has type t and gives no type binding.
- (15): if *cid* has type scheme $\forall. \tau \to t$ in Γ for type name t, and *pat* has type τ and introduces Γ', then *cid pat* has type t and introduces Γ'.
- (16): if pat_i has type τ_i and introduces Γ_i for $i = 1, 2$, then (pat_1, pat_2) has type $\tau_1 \times \tau_2$ and introduces $\Gamma_1 \oplus \Gamma_2$. Note that Γ_1 and Γ_2 do not share the same variable identifier (Why?).
- (17): if *pat* has type τ and introduces Γ', and if *ty* has the same type τ, then $(pat : ty)$ has type τ and introduces Γ'.

Rules for Expressions. The following inference rules are for $\Gamma \vdash exp \succ \tau$.

$$\frac{}{\Gamma \vdash num \succ \mathsf{int}}\ (18) \qquad \frac{}{\Gamma \vdash () \succ \mathsf{unit}}\ (19) \qquad \frac{}{\Gamma \vdash \mathbf{false} \succ \mathsf{bool}}\ (20) \qquad \frac{}{\Gamma \vdash \mathbf{true} \succ \mathsf{bool}}\ (21)$$

$$\frac{}{\Gamma \vdash + \succ \mathsf{int} \times \mathsf{int} \to \mathsf{int}}\ (22) \qquad \frac{}{\Gamma \vdash - \succ \mathsf{int} \times \mathsf{int} \to \mathsf{int}}\ (23) \qquad \frac{}{\Gamma \vdash * \succ \mathsf{int} \times \mathsf{int} \to \mathsf{int}}\ (24)$$

$$\frac{}{\Gamma \vdash = \succ \mathsf{int} \times \mathsf{int} \to \mathsf{bool}}\ (25) \qquad \frac{}{\Gamma \vdash < \succ \mathsf{int} \times \mathsf{int} \to \mathsf{bool}}\ (26)$$

$$\frac{id : \forall \vec{\alpha}. \tau \in \Gamma \quad \vec{\beta}\ \mathsf{fresh}}{\Gamma \vdash id \succ [\vec{\beta}/\vec{\alpha}]\tau}\ (27) \qquad \frac{\Gamma \vdash mrule \succ \tau}{\Gamma \vdash \mathbf{fun}\ mrule \succ \tau}\ (28)$$

$$\frac{\Gamma \vdash exp \succ \tau' \quad \Gamma \vdash mlist \succ \tau' \to \tau}{\Gamma \vdash \mathbf{match}\ exp\ \mathbf{with}\ mlist \succ \tau}\ (29) \qquad \frac{\Gamma \vdash exp_1 \succ \tau' \to \tau \quad \Gamma \vdash exp_2 \succ \tau'}{\Gamma \vdash exp_1\ exp_2 \succ \tau}\ (30)$$

$$\frac{\Gamma \vdash exp \succ \tau}{\Gamma \vdash \mathbf{ref}\ exp \succ \mathsf{ref}\ \tau}\ (31) \qquad \frac{\Gamma \vdash exp \succ \mathsf{ref}\ \tau}{\Gamma \vdash !\, exp \succ \tau}\ (32) \qquad \frac{\Gamma \vdash exp_1 \succ \mathsf{ref}\ \tau \quad \Gamma \vdash exp_2 \succ \tau}{\Gamma \vdash (exp_1 := exp_2) \succ \mathsf{unit}}\ (33)$$

$$\frac{\Gamma \vdash exp_1 \succ \tau_1 \quad \Gamma \vdash exp_2 \succ \tau_2}{\Gamma \vdash (exp_1, exp_2) \succ \tau_1 \times \tau_2}\ (34) \qquad \frac{\Gamma \vdash exp \succ \tau \quad \Gamma \vdash ty \succ \tau}{\Gamma \vdash (exp : ty) \succ \tau}\ (35)$$

$$\frac{\Gamma \vdash pat \succ \tau', \Gamma' \quad \Gamma \vdash exp_1 \succ \tau' \quad \Gamma \oplus \mathrm{Closure}_{\Gamma}^{exp_1}(\Gamma') \vdash exp_2 \succ \tau}{\Gamma \vdash \mathbf{let}\ pat = exp_1\ \mathbf{in}\ exp_2 \succ \tau}\ (36)$$

$$\frac{\Gamma \vdash pat \succ \tau', \Gamma' \quad \Gamma \oplus \Gamma' \vdash exp_1 \succ \tau' \quad \Gamma \oplus \mathrm{Closure}_{\Gamma}^{exp_1}(\Gamma') \vdash exp_2 \succ \tau}{\Gamma \vdash \mathbf{let\ rec}\ pat = exp_1\ \mathbf{in}\ exp_2 \succ \tau}\ (37)$$

- (18)–(21): each special constant has the predefined constant type.
- (22)–(26): each primitive operation also has the predefined type.
- (27): if *id* (either a variable or a constructor) has type scheme $\forall \vec{\alpha}. \tau$ in Γ, then *id* has type $[\vec{\beta}/\vec{\alpha}]\tau$, (that is, *id* is instantiated for each occurrence), where all type variables in $\vec{\beta}$ are fresh. Note that $[\vec{\beta}/\vec{\alpha}]$ substitutes $\vec{\beta}$ for $\vec{\alpha}$: for $\vec{\alpha} = \{\alpha_1, \cdots, \alpha_n\}$ and $\vec{\beta} = \{\beta_1, \cdots, \beta_n\}$ with the same size n, $[\vec{\beta}/\vec{\alpha}]$ is the same as $[\beta_n/\alpha_n] \cdots [\beta_1/\alpha_1]$.
- (28): if *mrule* has type τ, then **fun** *mrule* has type τ.
- (29): if *exp* has type τ' and *mlist* has type $\tau' \to \tau$, then **match** *exp* **with** *mlist* has type τ.
- (30): if exp_1 has type $\tau' \to \tau$ and exp_2 has type τ' for the same τ', then $exp_1\ exp_2$ has type τ.
- (31): if *exp* has type τ, then **ref** *exp* has type $\mathsf{ref}\ \tau$.
- (32): if *exp* has type $\mathsf{ref}\ \tau$, then $!\,exp$ has type τ.

- (33): if exp_1 has type $\mathsf{ref}\ \tau$ and exp_2 has type τ, then $exp_1 := exp_2$ has type unit.
- (34): if exp_i has type τ_i for $i = 1, 2$, then (exp_1, exp_2) has type $\tau_1 \times \tau_2$.
- (35): if both exp and ty have the same type τ, then $exp : ty$ has type τ.
- (36): if pat has type τ' and introduces Γ', exp_1 has the same type τ', and exp_2 has type τ under $\Gamma \oplus \mathrm{Closure}_{\Gamma}^{exp_1}(\Gamma')$, then $\mathbf{let}\ pat = exp_1\ \mathbf{in}\ exp_2$ has type τ.
- (37): if pat has type τ' and introduces Γ', exp_1 has the same type τ' under $\Gamma \oplus \Gamma'$, and exp_2 has type τ under $\Gamma \oplus \mathrm{Closure}_{\Gamma}^{exp_1}(\Gamma')$, then $\mathbf{let\ rec}\ pat = exp_1\ \mathbf{in}\ exp_2$ has type τ.

$\mathrm{Closure}_{\Gamma}^{exp}(\Gamma')$ generalizes each type scheme in Γ', taking into account free type variables and value restriction. Each type binding $id : \forall\vec{\alpha}.\, \tau$ in Γ' is generalized in $\mathrm{Closure}_{\Gamma}^{exp}(\Gamma')$ to

$$id : \forall (ftv(\tau) \setminus ftv(\Gamma)) \cup \vec{\alpha}.\, \tau,$$

provided that exp is syntactically a value in TML, where $ftv(\tau)$ and $ftv(\Gamma)$ denote the set of free type variables in τ and Γ, respectively. If exp is not a syntactic value, $\mathrm{Closure}_{\Gamma}^{exp}(\Gamma') = \Gamma'$ (see "Value restriction" section in the course note). E.g., for a syntactic value v:

$$\begin{aligned}
\mathrm{Closure}_{\cdot}^{v}(\{x : \forall.\, \alpha\}) &= x : \forall\alpha.\, \alpha \\
\mathrm{Closure}_{z\,:\,\forall.\,\alpha}^{v}(\{x : \forall.\, \alpha\}) &= x : \forall.\, \alpha \\
\mathrm{Closure}_{z\,:\,\forall\alpha.\,\alpha}^{v}(\{x : \forall\alpha.\, \alpha\}) &= x : \forall\alpha.\, \alpha
\end{aligned}$$

See "Type reconstruction algorithm" section in the course note; it explains $Gen_{\Gamma}A$, which is a simplified version of $\mathrm{Closure}_{\Gamma}^{exp}(\Gamma')$.

Rules for Match Rules. The following inference rules are for $\Gamma \vdash mlist \succ \tau$.

$$\frac{\Gamma \vdash pat \succ \tau, \Gamma' \quad \Gamma \oplus \Gamma' \vdash exp \succ \tau'}{\Gamma \vdash pat\ \texttt{->}\ exp \succ \tau \to \tau'}\ (38) \qquad \frac{\Gamma \vdash mrule \succ \tau \quad \Gamma \vdash mlist \succ \tau}{\Gamma \vdash mrule \mid mlist \succ \tau}\ (39)$$

- (38): if pat has type τ and introduces type bindings in Γ', and if exp has type τ' under $\Gamma \oplus \Gamma'$, then $pat\ \texttt{->}\ exp$ has type $\tau \to \tau'$.
- (39): if $mrule$ has type τ and $mlist$ has the same type τ, then $mrule \mid mlist$ has type τ.

Rules for Declarations. The following rules are for $\Gamma \vdash dlist \succ \Gamma'$ and $\Gamma \vdash_t conbind \succ \Gamma'$.

$$\frac{t\ \mathsf{fresh} \quad \Gamma \oplus tycon \mapsto t \vdash_t conbind \succ \Gamma'}{\Gamma \vdash \mathbf{type}\ tycon = conbind \succ \Gamma' \oplus tycon \mapsto t}\ (40) \qquad \frac{\Gamma \vdash dec \succ \Gamma' \quad \Gamma \oplus \Gamma' \vdash dlist \succ \Gamma''}{\Gamma \vdash dec\ dlist \succ \Gamma' \oplus \Gamma''}\ (41)$$

$$\frac{}{\Gamma \vdash_t cid \succ cid : \forall.\, t}\ (42) \qquad \frac{\Gamma \vdash ty \succ \tau}{\Gamma \vdash_t cid\ \mathbf{of}\ ty \succ cid : \forall. \tau \to t}\ (43)$$

$$\frac{\Gamma \vdash_t conbinding \succ \Gamma' \quad \Gamma \vdash_t conbind \succ \Gamma''}{\Gamma \vdash_t conbinding \ \texttt{|}\ conbind \succ \Gamma' \oplus \Gamma''}\ (44)$$

- (40): if $conbind$ introduces Γ' with a fresh type name t under typing context $\Gamma \oplus tycon \mapsto t$, then $\mathbf{type}\ tycon = conbind$ introduces $\Gamma' \oplus tycon \mapsto t$.
- (41): if dec introduces Γ' under Γ, and $dlist$ introduces Γ'' under $\Gamma \oplus \Gamma'$, then $dec\ dlist$ introduces $\Gamma' \oplus \Gamma''$ under Γ.
- (42): if cid exists alone, then cid introduces itself as an identifier with type scheme $\forall.\, t$.
- (43): if ty has type τ, then $cid\ \mathbf{of}\ ty$ introduces cid as an identifier with type scheme $\forall.\, \tau \to t$.
- (44): if $conbinding$ introduces Γ' and $conbind$ introduces Γ'', then $conbinding\ \texttt{|}\ conbind$ introduces $\Gamma' \oplus \Gamma''$.

2.3 Type Reconstruction Algorithm

This section provides you with some hints about how to derive a type reconstruction algorithm from the static semantics. The static semantics states a set of constraints. For example, the following rule (30) in the static semantics says that exp_1 should have type $\tau' \to \tau$ and exp_2 should have type τ' for the same type τ'. A type reconstruction algorithm should reconstruct a type which satisfies these constraints in the static semantics.

$$\frac{\Gamma \vdash exp_1 \succ \tau' \to \tau \quad \Gamma \vdash exp_2 \succ \tau'}{\Gamma \vdash exp_1\ exp_2 \succ \tau} \ (30)$$

How can we reconstruct a type that satisfies the constraints? As an example, we see how the algorithm W discussed in class reconstructs a type for $e_1\ e_2$. The typing rule for $e_1\ e_2$ is:

$$\frac{\Gamma \rhd e_1 : A \to B \quad \Gamma \rhd e_2 : A}{\Gamma \rhd e_1\ e_2 : B}$$

See "Implicit polymorphism" section in the course note for details. The constraint is that e_1 has type $A \to B$ and e_2 has type A for the same type A.

The algorithm W then reconstructs a type for $e_1\ e_2$ as follows. (See "Type reconstruction algorithm" section in the course note for more details.)

$$\begin{array}{rcll} W(\Gamma, e_1\ e_2) & = & \textbf{let } (S_1, A_1) = W(\Gamma, e_1) \textbf{ in} & \\ & & \textbf{let } (S_2, A_2) = W(S_1 \cdot \Gamma, e_2) \textbf{ in} & \\ & & \textbf{let } S_3 = \mathit{Unify}(S_2 \cdot A_1 = A_2 \to \alpha) \textbf{ in} & \quad \alpha \text{ fresh} \\ & & (S_3 \circ S_2 \circ S_1,\ S_3 \cdot \alpha) & \end{array}$$

The algorithm W infers type A_1 with type substitution S_1 under typing context Γ from e_1, and infers type A_2 with type substitution S_2 under typing context $S_1 \cdot \Gamma$ from e_2. Type substitution S_1 satisfies the constraints in e_1, and type substitution S_2 satisfies the constraints in e_2. The algorithm then runs $\mathit{Unify}(S_2 \cdot A_1 = A_2 \to \alpha)$ to check whether e_1 can have a function type whose argument type is the same with the type of e_2 and to obtain type substitution S_3 that satisfies this constraint if possible. (Note that $S_2 \cdot A_1$ denotes the type of e_1 when the constraints in e_2 are satisfied.) Finally, the algorithm returns $S_3 \circ S_2 \circ S_1$ that satisfies all the constraints in $e_1\ e_2$ and $S_3 \cdot \alpha$ that is the type of $e_1\ e_2$ when all the constraints are satisfied.

3 Programming Instruction

Download the zip file `hw9.zip` from the course webpage or by running `receive` on the server `141.223.163.223`, and unzip it on your working directory.

The goal is to implement the function `tprogram` in `hw9.ml`, which translates a well typed `Tml.program` into a `Typed.program`, where all expressions and patterns are annotated with types. If the `Tml.program` value does not typecheck, `tprogram` should raise `TypingError`.

```
exception NotImplemented
exception TypingError

(* tprogram : Tml.program -> Typed.program *)
let tprogram (dlist, exp) = raise NotImplemented
```

Note that all we care about is your implementation for `tprogram`. You can introduce new functions, new structures, new functors or whatever you consider to be instrumental to your implementation. However, place all new ones in the same file `hw9.ml`.

The file `typed.ml` defines types and datatypes for some semantic objects, besides types for annotated programs. The representation for each semantic object in `typed.ml` is as follows:

- tyvar α: type `tyvar`
- tyname t: type `tyname`
- semantic type τ: type `ty` (with `TINT`, `TBOOL`, `TUNIT`, `TNAME`, `TPAIR`, `TFUN`, `TREF`, `TVAR`).

For details of these types, please see `typed.ml`. You can also use the structure `Typedprint` defined in `typedprint.ml` to print the content of values in `Typed`, e.g., when debugging your code. The meaning of each function in the structure should be clear. For example, `exp2str exp` returns a string representation of the `Typed.exp` value `exp`.

4 Testing and Submission

After implementing the function `tprogram`, run the command "`ocamlbuild main.native`" to compile the sources files. Now you can typecheck TML programs in the `example` directory (or your own TML program in a file) using the executable `main.native`. For example:

```
$ ./main.native example/let.tml
***** Input program *****
(let x = 1 in
(+ (x, 1)))

***** Typed program *****
((let (x : int) = (1 : int) in
(((+ : ((int, int) -> int)) (((x : int), (1 : int)) : (int, int))) : int)) :
   int)

$ ./main.native example/poly.tml
***** Input program *****
(fun f -> (fun g -> (fun x -> (g (f x)))))

***** Typed program *****
((fun (f : ('3 -> '4)) -> ((fun (g : ('4 -> '5)) -> ((fun (x : '3) -> (((g :
   ('4 -> '5)) (((f : ('3 -> '4)) (x : '3)) : '4)) : '5)) : ('3 -> '5))) :
   (('4 -> '5) -> ('3 -> '5)))) : (('3 -> '4) -> (('4 -> '5) -> ('3 -> '5))))

$ ./main.native example/failure.tml
***** Input program *****
type t = T1
type t = T2
;;
(match 1 with 0 -> T1 | _ -> T2)

Fatal error: exception Hw9.TypingError
```

Make sure that you can compile `hw9.ml` by running `ocamlbuild main.native`. When you have the file `hw9.ml` ready for submission, run the `handin` command in the same directory, and your file will be submitted automatically. Good luck!