**The Type Inference Algorithm**

**∀-elimination:**
\[
\frac{x : \forall a_1 \ldots \forall a_n. \tau \in \Gamma}{\Gamma \vdash x : \beta} \quad \beta, \text{ fresh}
\]

**Application:**

**Slide 1**
\[
\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma' \vdash e_2 : \tau_2 \quad \Gamma' \vdash U^{\alpha} : \tau_2 \rightarrow \alpha}{\Gamma \vdash \text{apply}(e_1, e_2) : U^{\alpha} \quad \alpha \text{ fresh}}
\]

**Function definition:**
\[
\frac{\Gamma \vdash \{ x : \alpha \} \vdash e : \tau \quad \alpha \text{ fresh}}{\Gamma \vdash \text{letfun}(f.x.e) : T \alpha \rightarrow \tau}
\]

But wait!!! This only works for non-recursive functions!

**Function definition:**
\[
\frac{\Gamma \vdash \{ x : \alpha \} \cup \{ f : \alpha_2 \} \vdash e : \tau \quad \alpha_2 \text{ fresh}}{U \Gamma \vdash \text{letfun}(f.x.e) : T \alpha_1 \rightarrow \tau}
\]

**Slide 2**

What is the type of \( \text{letfun } f \ x = f \ x \)?
Specific rules for plus, mult, inl, can be derived from their type is in \( \Gamma \) and the application rule.

**Example:**

\[
\text{inl} : \forall a, \forall b, a \rightarrow (a + b)
\]

What is the type of \( \text{inl}(e) \) for an arbitrary expression \( e \)?

**Type Inference**

- the inference rules describe Robin Milner’s type inference algorithm \( W \)
- returns the same typing scheme as the non-syntax directed rules discussed previously (modulo \( \forall \)-quantification)

**Unification**

Simple unification algorithm:

- input: two type terms \( t_1 \) and \( t_2 \), forall quantified variables replaced by fresh, unique variables
- output: the most general unifier of \( t_1 \) and \( t_2 \) (if it exists)

**Cases:** \( t_1 \) and \( t_2 \)

1. both are type variables \( v_1 \) and \( v_2 \):
   - if \( v_1 = v_2 \), return the empty substitution
   - otherwise, return \( [v_1/v_2] \)
2. both are primitive types
   - if they are the same, return the empty substitution
   - otherwise, there is no unifier
3. both are product types, with \( t_1 = (t_{11} \times t_{12}) \), \( t_2 = (t_{21} \times t_{22}) \)
   - compute the mgu \( S \) of \( t_{11} \) and \( t_{21} \)
   - compute the mgu of \( S' \) of \( S \) \( t_{12} \) and \( S t_{22} \)
   - return \( S' \cup S \)
4. function types / sum types (see product types)
5. only one is a type variable \( v \), the other an arbitrary type term \( t \)
   - if \( v \) occurs in \( t \), there is no unifier
   - otherwise, return \( [t/v] \)
6. otherwise, there is no unifier
ASSIGNMENT 2

Implement type inference for MinHs:
- unification
- data type for substitution and operations on this type (see inference rules)
- free variable check
- type inference algorithm

data Bind = Bind Id (Maybe Type) [Id] Exp
    deriving (Read,Show,Eq)

tyInfBnd :: TypeEnv -> Bind -> TC (Bind, Type, Subst)
tyInfExpr :: TypeEnv -> Exp -> TC (Exp, Type, Subst)

Representation of Types:

<table>
<thead>
<tr>
<th>Type Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
</tr>
<tr>
<td>TyVarTy TyVar</td>
</tr>
<tr>
<td>FunTy Type Type</td>
</tr>
<tr>
<td>ForallTy TyVar Type -- a polymorphic type</td>
</tr>
<tr>
<td>TyApp Type Type     -- application</td>
</tr>
<tr>
<td>TyConstr TyCon     -- regular type constructor</td>
</tr>
</tbody>
</table>

data TyCon
    = UnitCon
    | BoolCon
    | IntCon
    | PairCon
    | SumCon

Some Examples:
(we represent terms of type Id as string here, in reality, the representation is more complicated but irrelevant for the assignment)

<table>
<thead>
<tr>
<th>Type</th>
<th>Term Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>a -&gt; b</td>
<td>FunTy (TyVarTy &quot;a&quot;) (TyVarTy &quot;b&quot;)</td>
</tr>
<tr>
<td>forall a a -&gt; a</td>
<td>Forall &quot;a&quot; (FunTy (TyVarTy &quot;a&quot;) (TyVarTy &quot;a&quot;))</td>
</tr>
<tr>
<td>Int</td>
<td>TyConstr IntCon</td>
</tr>
<tr>
<td>(Int, Bool)</td>
<td>TyApp (TyApp (TyConstr PairCon) (TyConstr IntCon)) (TyConstr BoolCon)</td>
</tr>
</tbody>
</table>
The TC Monad

- We need a supply of fresh names
- Have to keep track of the names already used
- Haskell does not have global variables or a global state

The function `freshName :: TC Id` returns a new identifier, wrapped in the `TC` type (compare to `Maybe` type).

Slide 13

There is only one way (you should use) to access the "contents" of a `TC` type, the `<-` operator:

- `new_id <- freshName` unwarps the `TC` type returned by `freshName` and binds the content to `new_id`.

Note that the type of the whole expression `new_id <- freshName` is again `TC Id`.

More operations and notation:

- `return :: a -> TC a` to wrap a value of type `a` into `TC a`.
- `let x = expr` when you want to bind a value of non-TC type `a` to a variable. This is different to usual let-bindings: if `expr` has type `a`, `let x = expr` has type `TC a`.
- The `do`-notation allows to evaluate a sequence of expressions of type `TC a`:

```
foo :: TC Int
foo =
do
  new_id <- freshName  -- :: TC Id
  let x = 5 * 3        -- :: TC Int
  return x            -- :: TC Int
```

Note that, given these operations, once we call `freshName` anywhere in a function, the whole function will have the result type `TC`, as there is no way to get rid of the `TC` constructor.

Example:

We implement the `forall` elimination rule:

\[
\frac{x : \forall a_1 \ldots \forall a_n, \pi \in \Gamma}{\Gamma \vdash x : [\beta_1/\alpha_1] \ldots [\beta_n/\alpha_n] \rho}, \beta, \text{fresh}
\]

which takes a type term, removes all `forall`-quantifiers, and replaces all the bound variables by fresh variables.

If type starts with `ForallTy`:

1. generate fresh type variable
2. replace all occurrences of bound variable with the new variable using substitution
3. call `forallElim` recursively on the result in case there are more quantifiers
4. return result

Otherwise, simply return original type: there is nothing.
Definition (type annotation of each expression in comments):

forallElim:: Type -> TC Type
forallElim (ForallTy i t) =
  do
    newId <- freshName -- :: TC Id
    let newT = applySubst [(i, TyVarTy newId)] t -- :: TC Type
    let resultT = forallElim newT -- :: TC Type
    return resultT -- :: TC Type

forallElim t = return t

Assuming that a substitution is defined to be list of (Id, Type),
and the function applySub:: Subst -> Type -> Type
applies a substitution to a type

Binding the result of the recursive call to resultT is not
necessary, we could just write:

forallElim:: Type -> TC Type
forallElim (ForallTy i t) =
  do
    newId <- freshName -- :: TC Id
    let newT = applySubst [(i, TyVarTy newId)] t -- :: TC Type
    forallElim newT -- :: TC Type
forallElim t = return t

TYPE CLASSES AND OVERLOADING

- We add the type Float to MinHs
- How does this affect the type of the built-in arithmetic
  operations?

Idea:

- Group types together which share some properties and
  operations into a class of types
  - Num denotes the class of numerical types which work with
    arithmetic operations
  - Eq is the class of types whose elements can be compared
    using ==

We write

- Num i to indicate that a type i is a member of the type class Num, and
- f :: Num i -> t to say that f has the type t under the condition
  that i is a member of the type class Num
Predicates \( \pi \ ::= \ D \tau \)

Polytypes \( \sigma \ ::= \pi \Rightarrow \sigma \mid \tau \mid \forall \ t. \sigma \)

Monotypes \( \tau \ ::= \ t \mid \ldots \)

Expressions \( \ e \ ::= \ \text{Fun}\ t\ \in\ e\mid \text{inst}(e, \tau)\mid \ldots \)

Values \( \ v \ ::= \ \text{Fun}\ t\ \in\ e\mid \ldots \)

where \( D \) are class names

New type of operations:

\[
\rightarrow (\ast) \ ::= \ \forall\ a.\ \text{Num}\ a \Rightarrow \ a \rightarrow a \rightarrow a \\
\rightarrow \ldots \ \\
\rightarrow (==) \ ::= \ \forall\ a.\ \text{Eq}\ a \Rightarrow \ a \rightarrow a \rightarrow \text{Bool}
\]

Note that

\[
1.0 + 1
\]

is not possible since addition requires both arguments to be of the same type!

For type inference, we need to know which types are in which class:

\[
\rightarrow \text{Num}\ \text{Int} \\
\rightarrow \text{Num}\ \text{Float} \\
\rightarrow \text{Eq}\ \text{Int} \\
\rightarrow \text{Eq}\ \text{Float} \\
\rightarrow \text{Eq}\ \text{Bool}
\]

Let \( P \) be the set of predicates

\[
\{\text{Num}\ \text{Int}, \ldots \forall\ a.\ \forall\ b.\ \text{Eq}\ \ a \Rightarrow \text{Eq}\ b \Rightarrow \text{Eq}(a, b)\}
\]

INFERRING PREDICATES

Given a predicate set \( P \), we say \( P \) entails a constraint \( c \)
(written \( P \vdash c \)) if and only if

1. \( c \in P \), or
2. \( P \vdash \forall\ a.\ c' \) and \( c = [t/a]c' \), or
3. \( P \vdash \pi \Rightarrow c \) and \( P \vdash \pi \)
TYPE INFERENCE

Previous rules stay as they are, we just add $P$

\[
\frac{x : \tau \in \Gamma}{P \vdash x : \tau} \quad \frac{P \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{P \vdash \text{apply}(e_1, e_2) : \tau_3}
\]

Slide 25

\[
\frac{P \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{P \vdash \text{pair}(e_1, e_2) : (\tau_1 \times \tau_2)}
\]

\[
\frac{P \vdash e : \forall \tau.\tau}{\Gamma \vdash e : [\tau_1/\tau]}\quad \frac{P \vdash e : \forall \tau.\tau}{\Gamma \vdash e : \forall \tau.\tau}
\]

We need two additional rules:

\rightarrow \Rightarrow -\text{ Elimination:}

\[
\frac{P \vdash e : \pi \Rightarrow \rho \quad P \vdash \pi}{P \vdash e : \rho}
\]

Slide 26

\rightarrow \Rightarrow -\text{- Introduction:}

\[
\frac{P, \pi \vdash e : \rho}{P \vdash e : \pi \Rightarrow \rho}
\]

Slide 27

Let $\{ (+) : \forall \ a. \ Num \ a \Rightarrow a \rightarrow a \} \subset \Gamma$ and $\{ \text{Num Int} \} \subset P$.

The $\Rightarrow -$ Elimination rule is, for example, necessary to infer

\[
\text{plus} :: \text{Float} \rightarrow \text{Float} \rightarrow \text{Float};
\]

1. $\text{plus} : \forall a. \text{Num} \ a \Rightarrow a \rightarrow a \rightarrow a$ (since this type is in $\Gamma$, this implies
2. $\text{plus} : \text{Num Float} \Rightarrow \text{Float} \rightarrow \text{Float}$ ($\Rightarrow -$ elimination rule), this implies
3. $\text{plus} :: \text{Float} \rightarrow \text{Float} \rightarrow \text{Float}$ ($\Rightarrow -$ elimination rule, since $P \vdash \text{Num Float}$)