Type Systems

Functional Programming SS2007

Stefan Blom

Revision: 1.9

1 Type Checking

In this section, we explain how to prove that a certain expression has a certain type. First, we need to fix what types are.

1.1 Types

Types are built from a set of *type variables* TVar (typical element $\alpha, \beta, \alpha_1, \alpha', \cdots$.)

Definition 1.1 The set of types Type is the least set such that

- For any *type variable* α, we have that α is a type.
- Given types τ_1, τ_2, the *function type* or *arrow type* $\tau_1 \to \tau_2$ is a type.
- If t is a *type constructor* with arity n and $\tau_1, \cdots, \tau_n$ are types then $t(\tau_1, \cdots, \tau_n)$ is a type.

The arrow is *right-associative*. That is,

$$\alpha \to \beta \to \gamma = \alpha \to (\beta \to \gamma)$$

Also, for some proofs and constructions we view the arrow type as a binary type constructor:

$$\tau_1 \to \tau_2 = \to (\tau_1, \tau_2)$$

A type is typically denoted with a τ with annotations (superscripts, subscripts, primes, etc.) For unknown type constructors we use c, f, g, t with various annotations. Type constructors, like function sysmbols have an arity. Examples of type constructors of arity 0 are bool, int and char. An example of a type constructor of arity 1 is list.

1.2 lambda calculus

The full OCaml syntax is rather extensive, so we start with its core the lambda calculus first. A lambda calculus expression is built from variables, application and abstraction:

$$M, N ::= x \mid M\,N \mid \lambda x.M$$

A variable by itself is unknown, so we must assume some type for it. The assumption are stored in a *type environment* A. A type environment is a comma separated list of type assumptions of the form $x : \tau$. For example: $x : \mathsf{int}, y : \mathsf{int}$. A *typing judgement* is a statement $A \vdash e : \tau$. The meaning of this statement is that given variables of the types mentioned in the environment A, the expression e if provably of type τ.

To find variables in the environment, we view A as a partial function:

$$A(x) = \begin{cases} \tau & \text{, if } A = A', x : \tau \\ A'(x) & \text{, if } A = A', y : \tau, x \neq y \\ \bot & \text{, otherwise} \end{cases}$$

For example, $(\underbrace{x : \alpha, y : \beta}_{A})(x) = \alpha, A(y) = \beta, A(z) = \bot$. With this partial function, we can define the typing rule for variables:

$$\frac{A(x) \neq \bot}{A, x : \tau \vdash x : A(x)}$$

Meaning that is the type of x is defined in A then we can derive that x is of the given type.

An application $M\,N$ is of type τ, if N is of a certain type τ' and M is a function which takes arguments of type τ' and has results of type τ. Formally:

$$\frac{A \vdash M : \tau' \to \tau \quad A \vdash N : \tau'}{A \vdash M\,N : \tau}$$

For example, if we have a function plus that takes two integers and returns an integer and two integer x and y then $\text{plus}\,x\,y$ is of type integer. That is we want to derive:

$$\text{plus} : \mathsf{int} \to \mathsf{int} \to \mathsf{int}, x : \mathsf{int}, y : \mathsf{int} \vdash \text{plus}\,x\,y : \mathsf{int}$$

This is done by building the proof tree

$$\dfrac{\dfrac{\dfrac{A(\text{plus}) = \mathsf{int} \to \mathsf{int} \to \mathsf{int}}{A \vdash \text{plus} : \mathsf{int} \to \mathsf{int} \to \mathsf{int}} \quad \dfrac{A(x) = \mathsf{int}}{A \vdash x : \mathsf{int}}}{A \vdash \text{plus}\,x : \mathsf{int} \to \mathsf{int}} \quad \dfrac{A(y) = \mathsf{int}}{A \vdash y : \mathsf{int}}}{\underbrace{\text{plus} : \mathsf{int} \to \mathsf{int} \to \mathsf{int}, x : \mathsf{int}, y : \mathsf{int}}_{A} \vdash \text{plus}\,x\,y : \mathsf{int}}$$

An abstraction $\lambda x.M$ stands for a function that takes an argument x and evaluates M. The rule for typing this expression is

$$\frac{A, x : \tau \vdash M : \tau'}{A \vdash \lambda x.M : \tau \to \tau'}$$

So to prove that a function, taking an argument x of type τ returns a value of type τ', we must prove that M is of type τ' assuming that x is of type τ.

For example, $\lambda x.x$ can be proven to be of type $\alpha \to \alpha$:

$$\dfrac{\dfrac{}{x : \alpha \vdash x : \alpha}}{\vdash \lambda x.x : \alpha \to \alpha}$$

Similarly, we can derive that $\lambda x\, y.x$ is of type $\alpha \to \beta \to \alpha$:

$$\cfrac{\cfrac{\overline{x : \alpha, y : \beta \vdash x : \alpha}}{x : \alpha \vdash \lambda y.x : \beta \to \alpha}}{\vdash \lambda x\, y.x : \alpha \to \beta \to \alpha}$$

Exercise 1.1 Derive the following type judgements:

(a) $\mathrm{plus} : \mathsf{int} \to \mathsf{int} \to \mathsf{int} \vdash \lambda x.\mathrm{plus}\, x\, x : \mathsf{int} \to \mathsf{int}$

(b) $\vdash \lambda x\, y\, z.(x\, z)(y\, z) : (\alpha \to \beta \to \gamma) \to (\alpha \to \beta) \to \alpha \to \gamma$

(c) $\vdash \lambda f\, g\, x.(f(g\, x)) : (\alpha \to \beta) \to (\beta \to \gamma) \to (\alpha \to \gamma)$.

Hint: introducing shorthand by underbracing a type or type environment and assigning it a name makes proofs shorter and more readable.

1.3 extensions of lambda calculus

The typing rule for the let is

$$\frac{A \vdash M : \tau' \quad A, x : \tau' \vdash N : \tau}{A \vdash \mathsf{let}\ x = M\ \mathsf{in}\ N : \tau}$$

We can prove that $\mathsf{let}\ \mathrm{id} = \lambda x.x\ \mathsf{in}\ \mathrm{id}\,(\lambda x.x) : \alpha \to \alpha$ as follows:

$$\cfrac{\cfrac{\overline{x : \alpha \to \alpha \vdash x : \alpha \to \alpha}}{\vdash \lambda x.x : (\alpha \to \alpha) \to (\alpha \to \alpha)} \quad \cfrac{\overline{\mathrm{id} : \tau \vdash \mathrm{id} : \tau} \quad \cfrac{\overline{x : \alpha \vdash x : \alpha}}{\vdash \lambda x.x : \alpha \to \alpha}}{\mathrm{id} : \underbrace{(\alpha \to \alpha) \to (\alpha \to \alpha)}_{\tau} \vdash \mathrm{id}\,(\lambda x.x) : \alpha \to \alpha}}{\vdash \mathsf{let}\ \mathrm{id} = \lambda x.x\ \mathsf{in}\ \mathrm{id}\,(\lambda x.x) : \alpha \to \alpha}$$

1.4 A subset of ocaml

The type system in the previous section allows us to prove types for several constructs in OCaml, but it is not powerfull enough yet. For example, there is no type τ such that we can derive $\vdash \mathsf{let}\ \mathrm{id} = \lambda x.x\ \mathsf{in}\ \mathrm{id}\,\mathrm{id} : \tau$.

The problem is that when we derive a type for the let, we must fix the type of id. Once the type is fixed we cannot apply id to itself anymore. This is different from OCaml, where if we define **let** id x = x;; we can then legally write write id id. The difference is that we have a type system that can only deal with *concrete types*, whereas the type system of OCaml also has *universal types* or *polymorphic types*. That is, if we write **let** id x = x;; then we define a variable id of type $\alpha \to \alpha$. (The `'a` in OCaml corresponds to α.) This must be read as for any replacement τ of α, we have that $\mathrm{id} : \tau \to \tau$.

However, if we were to allow this in general then the resulting type inference problem would be undecidable. Thus, we will treat pre-defined symbols different the symbols being defined. For

the pre-defined symbols, we use universal types. For the symbols being defined, we use concreate types.

Another feature of OCaml is that we can define types. The most important of these is the algebraic type. If we define an algebraic type then we add a type constructor and several (term) constructors. For example, defining

```
type 'a tree = Leaf | Node of 'a tree * 'a * 'a tree ;;
```

add a unary type constructor tree, a constant Leaf of type $\mathsf{tree}(\alpha)$ and a ternary constructor Node of type $\mathsf{tree}(\alpha) * \alpha * \mathsf{tree}(\alpha) \to \mathsf{tree}(\alpha)$.

To be able to deal with these definitions of types and functions, we view the previously defined environment as a local part and add a global part E to our environment. This global part consists of two environments $E = (C; F)$, where:

$$\begin{array}{ll} C = c_1 : \tau_1^c, \cdots, c_l : \tau_l^c & \text{Constructors} \\ F = f_1 : \tau_1, \cdots, f_n : \tau_n & \text{defined Function symbols} \end{array}$$

the possible types for constructors $(\tau_1^c, \cdots, \tau_l^c)$ are limited to types of the form $t(\vec{\alpha})$ or $\tau_1 * \cdots * \tau_n \to t(\vec{\alpha})$ for $n > 0$. Note that the arity of a constructor can always be deduced from the type.

To express this, we will now have judgements of the form

$$E; A \vdash e : \tau$$

where E is a global environment and A is a local environment. Looking up variables, is now a two-step procedure. First, we look in the local environment. Second, we look in the function part of the global environment. The first step is covered by the rule

$$\frac{A(x) \neq \perp}{E; A \vdash x : A(x)}$$

The second step is covered by a new rule:

$$\frac{A(x) = \perp \; F(x) \neq \perp \; \sigma : \mathsf{TVar} \to \mathsf{Type}}{(C; F); A \vdash x : F(x)\sigma}$$

The first two conditions ensure that x is defined in F but not in A. The third condition is not really a condition, but more a quantifying statement meaning that for any substitution σ the rule applies. By applying the substitution σ to the universal type of x, we *instantiate* the type of x.

With these rules, we have that

$$\frac{\epsilon(\mathrm{id}) = \perp \; (\mathrm{id} : \alpha \to \alpha)(\mathrm{id}) = \alpha \to \alpha \sigma : x \mapsto x}{(C; \mathrm{id} : \alpha \to \alpha); \epsilon \vdash \mathrm{id} : \alpha \to \alpha}$$

and

$$\frac{\epsilon(\mathrm{id}) = \perp \; (\mathrm{id} : \alpha \to \alpha)(\mathrm{id}) = \alpha \to \alpha \sigma = [\alpha := \alpha \to \alpha]}{(C; \mathrm{id} : \alpha \to \alpha); \epsilon \vdash \mathrm{id} : (\alpha \to \alpha) \to (\alpha \to \alpha)}$$

The motivation for these steps is too long for convenience, the only essential information is the substitution. If there is no confusion possible we can omit it:

$$\frac{}{(C; \mathrm{id} : \alpha \to \alpha); \epsilon \vdash \mathrm{id} : \alpha \to \alpha}$$

Otherwise, we can write it like:

$$\frac{[\alpha := \alpha \to \alpha]}{(C; \mathrm{id} : \alpha \to \alpha); \epsilon \vdash \mathrm{id} : (\alpha \to \alpha) \to (\alpha \to \alpha)}$$

The proof rules for application and abstraction are adapted as follows:

$$\frac{E; A \vdash M : \tau' \to \tau \quad E; A \vdash N : \tau'}{E; A \vdash M\, N : \tau} \quad \frac{E; (A, x : \tau) \vdash M : \tau'}{E; A \vdash \lambda x.M : \tau \to \tau'}$$

The rule for the let becomes:

$$\frac{E; A \vdash M : \tau' \quad E; (A, x : \tau') \vdash N : \tau}{E; A \vdash \mathsf{let}\ x = M\ \mathsf{in}\ N : \tau}$$

To deal with definitions at top level, we consider a program to be a list of type definitions, function declarations and expressions. At the top level, the local environement is always empty so we omit it. This gives us four rules:

$$\frac{}{(C; F) \vdash \epsilon}$$

The emtpy program can always be typed.

$$\frac{(C; F); f : \tau \vdash M : \tau (C; F, f : \tau) \vdash P}{(C; F) \vdash \mathsf{letrec}\ f = M\ ;; P}$$

A program starting with a recursive function can be typed if we can type the right-hand side of the definition with a type τ while locally assuming that the function being defined is also of type τ and we can type the remainder of the program while globally assuming that the function is of type τ.

$$\frac{(C; F); \epsilon \vdash M : \tau (C; F, f : \tau) \vdash P}{(C; F) \vdash \mathsf{letrec}\ M\ ;; P}$$

A program starting with an expression can be typed if both the expression and the remainder can be typed.

$$\frac{(C, c_1 : t(\vec{\alpha}), \cdots, c_m : t(\vec{\alpha}), d_1 : \tau_1 \to t(\vec{\alpha}), \cdots, d_n : \tau_n \to t(\vec{\alpha}); F) \vdash P}{(C; F) \vdash \mathsf{type}\, t(\vec{\alpha}) = c_1 \mid \cdots \mid c_m \mid d_1\ \mathsf{of}\ \tau_1 \mid \cdots d_n\ \mathsf{of}\ \tau_n ;; P}$$

A program starting with a type definition can be typed if we can type the remainder, while adding the declaration.

Constructors in OCaml are symbols that have an arity, which can be derived from the type. If the type of a constructor c is $t(\vec{\alpha})$ then the arity of c is 0. If the type of c is of the form $\tau_1 * \cdots * \tau_n \to t(\vec{\alpha})$ then the arity of c is n, where if $n = 1$, we have that τ_1 is not a tuple.

We have two special cases of constructors: tuples and lists.

Tuples are embedded in OCaml, by using (type) constructors $*_n$ for $n \geq 2$. We have the abbreviations

$$\begin{array}{ll} *_n(\tau_1, \cdots, \tau_n) & = \tau_1 * \cdots * \tau_n \\ {}_n(M_1, \cdots, M_n) & = (M_1, \cdots, M_n) \end{array}$$

for type constructors and (expression) constructors, respectively.

Lists use a unary type constructor list E.g. $\mathsf{list}(\alpha)$ (written in ASCII as `'a list`), a constant [] and a binary constructor ::, written in infix notation.

The typing rules for constructors are:

$$\frac{C(c) = t(\vec{\alpha}) \quad \sigma : \mathsf{TVar} \to \mathsf{Type}}{(C;F); A \vdash c : t(\vec{\alpha})\sigma}$$

and

$$\frac{C(c) = \tau_1 * \cdots * \tau_n \to t(\vec{\alpha}) \quad \sigma : \mathsf{TVar} \to \mathsf{Type} \quad (C;F); A \vdash M_1 : \tau_1\sigma \cdots (C;F); A \vdash M_n : \tau_n\sigma}{(C;F); A \vdash c(M_1, \cdots, M_n) : t(\vec{\alpha})\sigma}$$

What we will do next is a give typing rule for the match expression

$$\mathsf{match}\, M\, \mathsf{with} |\ p_1\ \mathsf{when}\ c_1 \to N_1 \cdots |\ p_n\ \mathsf{when}\ c_n \to N_n$$

To prove this match statement of type τ, we must prove M to be of a chosen type τ' first and then for each of the cases, we must choose types for the variables in the pattern A_i and then for that choice prove that the pattern p_i has type τ', the condition c_i has type bool and the expression N_i has type τ.

$$\frac{E; A \vdash M : \tau' \quad \text{for } i = 1 \cdots n : p_i\ GV\ A_i \quad E; A, A_i \vdash p_i : \tau' \quad E; A, A_i \vdash c_i : \mathsf{bool} \quad E; A, A_i \vdash N_i : \tau}{E; A \vdash \mathsf{match}\, M\, \mathsf{with} |\ p_1\ \mathsf{when}\ c_1 \to N_1 \cdots |\ p_n\ \mathsf{when}\ c_n \to N_n : tau}$$

where

$$\frac{}{x\ GV\ x : \tau} \qquad \frac{p_1\ GV\ A_1 \quad p_n\ GV\ A_n}{t(p_1, \cdots, p_n)\ GV\ A_1, \cdots, A_n}$$

2 Type Inference

In the previous section, we have shown how to prove that $\lambda x\, y.x$ is of type $\alpha \to \beta \to \gamma$. In this section, we will concern ourselves with the task of *inferring* the type of a given expression.

Type inference is based on the idea that we build a set of equations in such a way that if we solve the set of equations, we have a proof of the type of the expression. For example: if we want to prove that

$$\vdash \lambda x\, y.x : \alpha_1$$

Then we need to show that

$$x : \alpha_2 \vdash \lambda y.x : \alpha_3 \text{ for } \alpha_1 \approx \alpha_2 \to \alpha_3$$

In turn this requires that we prove

$$x : \alpha_2, y : \alpha_4 \vdash x : \alpha_5 \text{ for } \alpha_3 \approx \alpha_4 \to \alpha_5$$

Which can be proven if $\alpha_2 \approx \alpha_5$. In tree form this reads as:

$$\frac{\dfrac{\dfrac{\alpha_2 = \alpha_5}{x : \alpha_2, y : \alpha_4 \vdash x : \alpha_5} \quad \alpha_3 = \alpha_4 \to \alpha_5}{x : \alpha_2 \vdash \lambda y.x : \alpha_3} \qquad \alpha_1 = \alpha_2 \to \alpha_3}{\vdash \lambda x\, y.x : \alpha_1}$$

So we need to solve the set of equations

$$\alpha_1 \approx \alpha_2 \to \alpha_3, \alpha_3 \approx \alpha_4 \to \alpha_5, \alpha_2 \approx \alpha_5$$

This task is known as *unification*.

2.1 Unification of Type Expressions

Unification is the task of finding a substitution, such that for a given set of equations applying the substitution to the left and right-hand sides yields that same type expression.

2.1.1 Problem Definition

Definition 2.1 Given a set of equation of type variables,

$$E = \{\tau_1 \approx \tau_1', \cdots, \tau_n \approx \tau_n'\}$$

a *unifier* of E is a substitition σ, such that

$$\tau_1\sigma = \tau_1'\sigma \wedge \cdots \wedge \tau_n\sigma = \tau_n'\sigma$$

Any given set of equations has many solutions. We say that one solution is more general than a second solution, if the second solution can be decomposed into the first solution plus some modifications.

Definition 2.2 Given a set of equations E. We say that a unifier σ_1 is *more general* than a unifier σ_2 (denoted $\sigma_1 \sqsubseteq \sigma_2$) if we can find a substitution σ_1' such that $\sigma_2 = \sigma_1\sigma_1'$.

The best possible solution to a set of equation is of course a solution which is more general than any other solution. Such a solution is known as a most general unifier.

Definition 2.3 Given a set of equations E. A unifier σ is a *most general unifier* if for all unifiers σ', we have $\sigma \sqsubset \sigma'$.

2.1.2 Substitutions

If σ_1 and σ_2 are substitutions then we denote their composition with as follows:

$$\sigma_1\sigma_2 \stackrel{\text{def}}{=} \sigma_2 \circ \sigma_1$$

Because function composition is associative, we get that substitution composition is associative as well:

$$\sigma_1(\sigma_2\sigma_3) = (\sigma_1\sigma_2)\sigma_3 \ .$$

The same holds for applying substitutions to terms:

$$(t\sigma_1)\sigma_2 = t(\sigma_1\sigma_2) \ .$$

Finite substitutions are denoted with a list of bindings:

$$[x_1 := t_1, \cdots, x_n := t_n] \stackrel{\text{def}}{=} x \mapsto \begin{cases} t_i & \text{, if } x = x_i \text{ for } i \in \{1, \cdots, n\} \\ x & \text{, otherwise} \end{cases}$$

```
function mgu(E) is
    if E = ∅ then return []; else
        take eq from E;
        if eq ≡ τ ≈ τ then return mgu(E); end;
        if eq ≡ α ≈ τ or eq ≡ τ ≈ α then
            if α occurs in τ then FAIL; end ;
            E := {u[α := τ] ≈ v[α := τ] | u ≈ v ∈ E};
            return [α := τ] mgu(E);
        end;
        if eq ≡ f(τ_1, ···, τ_n) ≈ g(τ'_1, ···, τ'_k) then
            if f ≠ g or n ≠ k then FAIL; end;
            for i in 1 to n do add τ_i ≈ τ'_i to E ; end;
            return mgu(E);
        end;
    end;
end mgu;
```

Table 1: Unification with recursion.

Proposition 2.4 Given two finite substitutions

$$\sigma_1 = [x_1 := t_1, \cdots, x_n := t_n] \text{ and } \sigma_2 = [y_1 := t'_1, \cdots, y_m := t'_m]$$

such that

$$\{y_1, \cdots, y_K\} \cap \{x_1, \cdots, x_n\} = \emptyset \text{ and } \{y_{K+1}, \cdots, y_m\} \subseteq \{x_1, \cdots, x_n\}$$

then

$$\sigma_1\sigma_2 = [x_1 := t_1\sigma_2, \cdots, x_n := t_n\sigma_2, y_1 := t'_1, \cdots, y_K := t'_K]$$

2.1.3 The Algorithm

Finding a most general unifier for a set of equations can be done recursively by analysing a given set of equations. If the set is empty then any substitution is a unifier and [] is a mgu. If the set is not empty then we pick one of the equations. If it is an equation of the form $\tau \approx \tau$ then any substitution is a unifier so we consider the remaining equations. If the equation of the for $\alpha \approx \tau$ or $\tau \approx\approx$ then if α occurs in τ, a unifier cannot exist. This can be seen by counting symbols. Suppose σ is a unifier of $\alpha \approx \tau$ and α occurs in τ then τ cannot be α because the first cause does not apply. So τ has at least two symbols, including an α that means that $\tau\sigma$ contains $\sigma(\alpha)$ and at least one more symbol. So $\tau\sigma$ has at least one more symbol than $\alpha\sigma$. If τ does not contain α then we must have that $\alpha = \tau$, so we apply the substitutions $[\alpha := \tau]$ to the remaining equations. The other case is $f(\tau_1, \cdots, \tau_n) \approx g(\tau'_1, \cdots, \tau'_k)$ If $f \neq g$ or $n \neq k$ then there does not exists a substitution that unifies this equation otherwise it depends if we can unify the argument, so we add $\tau_1 \approx \tau'_1, \cdots, \tau_n \approx \tau'_n$ to the set of equations. In table 1, this algorithm has been written out using recursion. Table 2 has a version that uses a while loop.

We continue by proving the correctness, but to do so we need two measures on sets of equations. By $|E|_{\text{vars}}$ we denote the number of different variables occurring in E. By $|E|_{\text{syms}}$ we denote the

number of symbols in the terms in E. If

$$E = \{\alpha \approx \beta \to \alpha\}$$

then $|E|_{\text{vars}} = 2$ (the variables are α and β) and $|E|_{\text{syms}} = 4$ (there is one α, two β's and one $\to$; the $\approx$ does not count).

Proposition 2.5 If mgu(E) returns a substitution then that substitution is a most general unifier.

Proof. We need to prove two statements:

(i) If mgu(E) returns a substitution then that substitution is a unifier of E.

(ii) If σ is a unifier of E then $\sigma \sqsubseteq$ mgu(E).

(i) Proof by induction on $(|E|_{\text{vars}}, |E|_{\text{syms}})$ using lexicographic ordering.
If $E = \emptyset$ then mgu(E)=[], which is a unifier of E.
If $E = \{eq\} \cup E'$ then we must distinguish cases for eq:

- If $eq \equiv \tau \approx \tau$ then any substitution is a unifier of { eq }. By induction hypothesis mgu(E) is a unifier of E'. If a substitution is a unifier of two sets of equation then it is a unifier for the union, so mgu(E) is a unifier of E.
- Consider the case $eq \equiv \alpha \approx \tau$ or $eq \equiv \tau \approx \alpha$. If α occurs in τ then mgu does not return a substitution, so α does not occur in τ. That means that $E'[\alpha := \tau]$ contains one variable less than E, so by induction hypothesis mgu($E'[\alpha := \tau]$) is a unifier of $E'[\alpha := \tau]$. Therefore $[\alpha := \tau]$mgu($E'[\alpha := \tau]$) is a unifier of E'. Because α does not occur in τ, we have that $[\alpha := \tau]$mgu($E'[\alpha := \tau]$) is a unifier of $\{eq\}$ so $[\alpha := \tau]$mgu($E'[\alpha := \tau]$) is a unifier of E.
- If $eq \equiv f(\tau_1, \cdots, \tau_n) \approx g(\tau'_1, \cdots, \tau'_k)$ and $f \neq g$ or $n \neq k$ then mgu(E) does not return a substitution so $f = g$ and $n = k$. Let $E'' = \{\tau_1 \approx \tau'_1, \cdots, \tau_n \approx \tau'_k\} \cup E'$. By induction hypothesis mgu(E'') is a unifier of E''. That means that

$$\begin{array}{lll} f(\tau_1, \cdots, \tau_n)\text{mgu(E}'') & = f(\tau_1\text{mgu(E}''), \cdots, \tau_n\text{mgu(E}'')) & \text{definition of substitution} \\ & = f(\tau'_1\text{mgu(E}''), \cdots, \tau'_n\text{mgu(E}'')) & \text{mgu}(E'') \text{ is a unifier of } E''. \\ & = g(\tau'_1\text{mgu(E}''), \cdots, \tau'_k\text{mgu(E}'')) & f = g, n = k \\ & = g(\tau'_1, \cdots, \tau'_k)\text{mgu(E}'') & \text{definition of substitution} \end{array}$$

  So, mgu(E'') is a unifier of { eq} and hence E.

(ii) Proof by induction on $(|E|_{\text{vars}}, |E|_{\text{syms}})$ using lexicographic ordering.
If $E = \emptyset$ then mgu(E) = [] $\sqsubseteq \sigma$.
If $E = \{eq\} \cup E'$ then we must distinguish cases for eq:

- If $eq \equiv t = t$ then σ is a unifier of E', so

$$\text{mgu}(E) = \text{mgu}(E') \overset{\text{IH}}{\sqsubseteq} \sigma$$

```
procedure mgu(E) is
    let σ = [];
    while E ≠ ∅ do
        take eq from E;
        if eq ≡ τ ≈ τ then next; end;
        if eq ≡ α ≈ τ or eq ≡ τ ≈ α then
            if α occurs in τ then FAIL ; end ;
            E := {u[α := τ] ≈ v[α := τ] | u ≈ v ∈ E};
            σ := σ[α := τ];
        end;
        if eq ≡ f(τ_1, ···, τ_n) ≈ g(τ'_1, ···, τ'_k) then
            if f ≠ g or n ≠ k then FAIL; end;
            for i in 1 to n do add τ_i ≈ τ'_i to E ; end;
        end;
    end;
    return σ;
end mgu;
```

Table 2: Unification with a while-loop.

- Consider the case $eq \equiv \alpha \approx \tau$ or $eq \equiv \tau \approx \alpha$. If α occurs in τ then mgu does not return a substitution, so α does not occur in τ. Because σ is a unifier of E, we have that $\alpha\sigma = \tau\sigma$. From this it follows that $\sigma = [\alpha := \tau]\sigma$. Hence $[\alpha := \tau]\sigma$ is a unifier of E', which means that σ is a unifier of $E'[\alpha := \tau]$. By induction hypothesis we get

$$(\text{mgu})(\text{E}'[\alpha := \tau]) \sqsubseteq \sigma$$

  If $\sigma_1 \sqsubseteq \sigma_2$ then $\sigma'\sigma_1 \sqsubseteq \sigma'\sigma_2$ so

$$(\text{mgu})(\text{E}) = [\alpha := \tau](\text{mgu})(\text{E}'[\alpha := \tau]) \sqsubseteq [\alpha := \tau]\sigma = \sigma$$

- If $eq \equiv f(\tau_1, \cdots, \tau_n) \approx g(\tau'_1, \cdots, \tau'_k)$ and $f \neq g$ or $n \neq k$ then mgu(E) does not return a substitution so $f = g$ and $n = k$. Let $E'' = \{\tau_1 \approx \tau'_1, \cdots, \tau_n \approx \tau'_k\} \cup E'$. If σ is a unifier of E then σ is a unifier of E'' as well so

$$(\text{mgu})(\text{E}) = (\text{mgu})(\text{E}'') \sqsubseteq \sigma$$

□

3 The type inference algorithm.

In this section, we present a type inference algorithm for recursive definitions

$$\mathsf{letrec}\ x = M\ ;;$$

The algorithm has three steps. First, we label the definition of M with type variables. Second, we extract a set of equations. Third, we solve the equations.

3.1 Examples

Consider the definition

$$\mathsf{letrec}\ id = \lambda x.x\ ;;$$

First, we label this definition:

$$\mathsf{letrec}\ id : \alpha = (\lambda x : \alpha.x : \alpha) : \beta\ ;;$$

That is, we attach a type variable to every sub-term and every binder (the x in $\lambda x.M$ or $\mathsf{let}\ x = M\ \mathsf{in}\ N$). In principle, every sub-term is labeled with a fresh variable. However, we make an exception for bound variables. Bound variables get the same label as the variable they are bound to. So $\lambda x.y\,x$ might be labeled as $(\lambda x : \alpha.(y : \beta)\,(x : \alpha)) : \gamma$, but not as $(\lambda x : \alpha.(y : \beta)\,(x : \delta)) : \gamma$.

The second step is to extract type equations. For the top level, we must have that the labels of the left and right-hand sides are identical:

$$\alpha \approx \beta$$

For every subterm of the form $(\lambda x : \alpha_1.M : \alpha_2) : \alpha_3$, we get an equation $\alpha_3 \approx \alpha_1 \to \alpha_3$:

$$\beta \approx \alpha \to \alpha$$

Subterms of the form $x : \alpha$ are correctly typed if the type of the variable is the same as the type of the binder, which is taken care of by definition.

This gives us the equations

$$\alpha \approx \beta, \beta \approx \alpha \to \alpha$$

A mgu for this set of equations is

$$[\alpha := \alpha \to \alpha, \beta := \alpha \to \alpha]$$

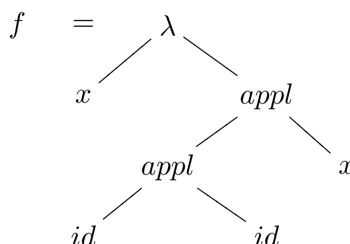
Consider the definition

$$\mathsf{letrec}\ f = \lambda x.id\,id\,x\ ;;$$

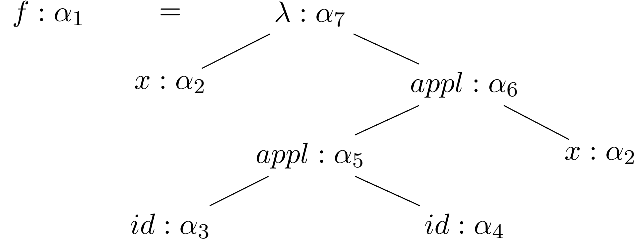
A labeled version of this definition is

$$\mathsf{letrec}\ f : \alpha_1 = (\lambda x : \alpha_2.(((id : \alpha_3)\,(id : \alpha_4)) : \alpha_5\,(x : \alpha_2)) : \alpha_6) : \alpha_7\ ;;$$

It is easier to see how the labeling occurs in a parse tree. The equation



is labeled as



Note how, the bound variable x gets the label α_2. Also note that the variable id, which was previously defined gets a different label for every occurrence on purpose.

The old cases for generating equations give us

$$\alpha_1 \approx \alpha_7, \alpha_7 \approx \alpha_2 \to \alpha_6$$

We have two new cases for generating equations.

- For every application $((M : \beta_1)(N : \beta_2)) : \beta_3$, we add $\beta_1 \approx \beta_2 \to \beta_3$. For our example this means
$$\alpha_5 \approx \alpha_2 \to \alpha_6, \alpha_3 \approx \alpha_4 \to \alpha_5$$
- For every occurrence of a previously defined variable $x : \alpha$, we must find the type τ of x; create a fresh instance τ' of x and add the equation $\alpha \approx \tau'$. Informally an instance is a renaming of a type, where every type variable is replaced by a fresh one. Thus, we need *two* fresh instances of $\alpha \to \alpha$. We take $\alpha_8 \to \alpha_8$ and $\alpha_9 \to \alpha_9$ and thus we get the equations
$$\alpha_3 \approx \alpha_8 \to \alpha_8, \alpha_4 \approx \alpha_9 \to \alpha_9$$

The resulting set of equations is

$$\{\alpha_1 \approx \alpha_7, \alpha_7 \approx \alpha_2 \to \alpha_6, \alpha_5 \approx \alpha_2 \to \alpha_6, \alpha_3 \approx \alpha_4 \to \alpha_5, \alpha_3 \approx \alpha_8 \to \alpha_8, \alpha_4 \approx \alpha_9 \to \alpha_9\}$$

For a computer, evaluating the mgu of such a tiny set of equations is easy. For a human, evaluating the mgu of a medium sized set of equations is already too much work without some speed-ups. We have a few speed-ups:

- If you are not interested in every variable, but just in γ then leave an equation $\gamma \approx \tau$ until the end. (Unless γ occurs in τ.)
- If you have equations
$$\underbrace{\alpha_1 \approx \tau_1, \cdots, \alpha_n \approx \tau_n}_{E}$$
such that α_i does not occur in τ_j for $i, j = 1 \cdots n$ then
$$\text{mgu}(E, E') = \underbrace{[\alpha_1 := \tau_1, \cdots, \alpha_n := \tau_n]}_{\sigma} \text{mgu}(E'\sigma)$$
- You can do more than one equation $f(\tau_1, \cdots, \tau_n) \approx g(\tau'_1, \cdots, \tau'_k)$ in one step.

If we underline the equations seleted then these speedups are applied as follows.

$$\{\alpha_1 \approx \alpha_7, \alpha_7 \approx \alpha_2 \to \alpha_6, \alpha_5 \approx \alpha_2 \to \alpha_6, \alpha_3 \approx \alpha_4 \to \alpha_5, \underline{\alpha_3 \approx \alpha_8 \to \alpha_8}, \underline{\alpha_4 \approx \alpha_9 \to \alpha_9}\}$$

Removing the selected equations and applying $[\alpha_3 := \alpha_8 \to \alpha_8, \alpha_4 := \alpha_9 \to \alpha_9]$ yields

$$\{\alpha_1 \approx \alpha_7, \underline{\alpha_7 \approx \alpha_2 \to \alpha_6}, \alpha_5 \approx \alpha_2 \to \alpha_6, \alpha_8 \to \alpha_8 \approx \alpha_9 \to \alpha_9 \to \alpha_5\}$$

remove and apply $[\alpha_7 := \alpha_2 \to \alpha_6]$

$$\{\alpha_1 \approx \alpha_2 \to \alpha_6, \underline{\alpha_5 \approx \alpha_2 \to \alpha_6}, \alpha_8 \to \alpha_8 \approx \alpha_9 \to \alpha_9 \to \alpha_5\}$$

$[\alpha_5 := \alpha_2 \to \alpha_6]$

$$\{\alpha_1 \approx \alpha_2 \to \alpha_6, \underline{\alpha_8 \to \alpha_8 \approx \alpha_9 \to \alpha_9 \to (\alpha_2 \to \alpha_6)}\}$$

simplifying

$$\{\alpha_1 \approx \alpha_2 \to \alpha_6, \underline{\alpha_8 \approx \alpha_9 \to \alpha_9}, \alpha_8 \approx \alpha_2 \to \alpha_6\}$$

$[\alpha_8 := \alpha_9 \to \alpha_9]$

$$\{\alpha_1 \approx \alpha_2 \to \alpha_6, \underline{\alpha_9 \to \alpha_9 \approx \alpha_2 \to \alpha_6}\}$$

$$\{\alpha_1 \approx \alpha_2 \to \alpha_6, \underline{\alpha_9 \approx \alpha_2}, \alpha_9 \approx \alpha_6\}$$

$[\alpha_9 := \alpha_2]$

$$\{\alpha_1 \approx \alpha_2 \to \alpha_6, \underline{\alpha_2 \approx \alpha_6}\}$$

$[\alpha_2 := \alpha_6]$

$$\{\alpha_1 \approx \alpha_6 \to \alpha_6\}$$

$[\alpha_1 := \alpha_6 \to \alpha_6]$

$$\emptyset$$

The mgu σ can be computed by evaluating

$\sigma = [\alpha_3 := \alpha_8 \to \alpha_8, \alpha_4 := \alpha_9 \to \alpha_9][\alpha_7 := \alpha_2 \to \alpha_6][\alpha_5 := \alpha_2 \to \alpha_6][\alpha_8 := \alpha_9 \to \alpha_9][\alpha_9 := \alpha_2][\alpha_2 := \alpha_6][\alpha_1 := \alpha_6$

But we do not have to do that. We only need $\sigma(\alpha_1)$, which is

$$\sigma(\alpha_1) = \alpha_6 \to \alpha_6$$

So the type of f is $\alpha_6 \to \alpha_6$.

If a function cannot be typed then the solving will fail. For example

$$\mathsf{letrec}\ w = \lambda x.x\,x\ ;;$$

Get labeled as

$$\mathsf{letrec}\ w : \alpha_1 = (\lambda x : \alpha_2.((x : \alpha_1)\,(x : \alpha_1)) : \alpha_3) : \alpha_4\ ;;$$

which gives rise to the equations

$$\alpha_1 \approx \alpha_4, \alpha_4 \approx \alpha_2 \to \alpha_3, \alpha_1 \approx \alpha_1 \to \alpha_3$$

which cannot be solved because α_1 occurs in $\alpha_1 \to \alpha_3 \neq \alpha_1$.

As a realistic example, let us consider

$$\mathsf{letrec}\ len = \mathsf{fun}\, l \rightarrow \mathsf{match}\, l\ \mathsf{with} |\ [] \rightarrow 0 \mid x :: xs \rightarrow 1 + (len\ xs)\ ;;$$

To make things more readable, we write the labels below the sub-expression rather than inside of it:

$$\mathsf{letrec}\ \underbrace{len}_{\alpha_1} = \underbrace{\mathsf{fun}\ \underbrace{l}_{\alpha_2} \rightarrow \underbrace{\mathsf{match}\ \underbrace{l}_{\alpha_2}\ \mathsf{with} |\ \underbrace{[]}_{\alpha_3} \rightarrow \underbrace{0}_{\alpha_4} \mid \underbrace{\underbrace{x}_{\alpha_5} :: \underbrace{xs}_{\alpha_6}}_{\alpha_7} \rightarrow \underbrace{\underbrace{1}_{\alpha_8} + \underbrace{(\underbrace{len}_{\alpha_1}\ \underbrace{xs}_{\alpha_6})}_{\alpha_9}}_{\alpha_{10}}}_{\alpha_{11}}}_{\alpha_{12}}\ ;;$$

Note that the occurrences of x and xs in the pattern $x :: xs$ are binding occurrences, so the bound occurrence of xs gets the same label.

For the definition we get

$$\alpha_1 \approx \alpha_{12}$$

For the function we get

$$\alpha_{12} \approx \alpha_2 \rightarrow \alpha_{11}$$

For the match, the patterns must have the same type as the expression doing the case distinction on:

$$\alpha_2 \approx \alpha_3, \alpha_2 \approx \alpha_7$$

and the results of each case must match the type of the match:

$$\alpha_4 \approx \alpha_{11}, \alpha_{10} \approx \alpha_{11}$$

For constructor, we must create instances as well. The constructor [] is of type $\mathsf{list}(\alpha)$, so we use $\mathsf{list}(\alpha_{13})$ as instance and add

$$\alpha_3 \approx \mathsf{list}(\alpha_{13})$$

0 is a constructor of type int, so

$$\alpha_4 \approx \mathsf{int}$$

:: is of arity 2 and type $\alpha * \mathsf{list}(\alpha) \rightarrow \mathsf{list}(\alpha)$. We use the instance $\alpha_{14} * \mathsf{list}(\alpha_{14}) \rightarrow \mathsf{list}(\alpha_{14})$ and get 3 equations

$$\alpha_5 \approx \alpha_{14}, \alpha_6 \approx \mathsf{list}(\alpha_{14}), \alpha_7 \approx \mathsf{list}(\alpha_{14})$$

$+$ is of type $\mathsf{int} \rightarrow \mathsf{int} \rightarrow \mathsf{int}$, which gives us:

$$\alpha_8 \approx \mathsf{int}, \alpha_9 \approx \mathsf{int}, \alpha_{10} \approx \mathsf{int}$$

The application gives

$$\alpha_1 \approx \alpha_6 \rightarrow \alpha_9$$

The whole set of equations is

$$\{\ \underline{\alpha_1 \approx \alpha_{12}}, \alpha_{12} \approx \alpha_2 \rightarrow \alpha_{11}, \alpha_2 \approx \alpha_3, \alpha_2 \approx \alpha_7, \alpha_4 \approx \alpha_{11}, \alpha_{10} \approx \alpha_{11}, \alpha_3 \approx \mathsf{list}(\alpha_{13}), \alpha_4 \approx \mathsf{int}, \alpha_5 \approx \alpha_{14}, \alpha_6 \approx \mathsf{list}(\alpha_{14}), \alpha_7 \approx \mathsf{list}(\alpha_{14}), \alpha_8 \approx \mathsf{int}, \alpha_9 \approx \mathsf{int}, \alpha_{10} \approx \mathsf{int}, \alpha_1 \approx \alpha_6 \rightarrow \alpha_9\}$$

$[\alpha_{12} := \alpha_1]$

$$\{\ \alpha_1 \approx \alpha_2 \to \alpha_{11}, \alpha_2 \approx \alpha_3, \alpha_2 \approx \alpha_7, \alpha_4 \approx \alpha_{11}, \alpha_{10} \approx \alpha_{11}, \underline{\alpha_3 \approx \mathsf{list}(\alpha_{13})}, \underline{\alpha_4 \approx \mathsf{int}}, \\ \alpha_5 \approx \alpha_{14}, \underline{\alpha_6 \approx \mathsf{list}(\alpha_{14})}, \underline{\alpha_7 \approx \mathsf{list}(\alpha_{14})}, \underline{\alpha_8 \approx \mathsf{int}}, \underline{\alpha_9 \approx \mathsf{int}}, \underline{\alpha_{10} \approx \mathsf{int}}, \alpha_1 \approx \alpha_6 \to \alpha_9\}$$

$[\alpha_3 := \mathsf{list}(\alpha_{13}), \alpha_4 := \mathsf{int}, \alpha_6 := \mathsf{list}(\alpha_{14}), \alpha_7 \approx \mathsf{list}(\alpha_{14}), \alpha_8 \approx \mathsf{int}, \alpha_9 \approx \mathsf{int}, \alpha_{10} \approx \mathsf{int}]$

$\{\ \alpha_1 \approx \alpha_2 \to \alpha_{11}, \alpha_2 \approx \mathsf{list}(\alpha_{13}), \underline{\alpha_2 \approx \mathsf{list}(\alpha_{14})}, [\mathsf{int} \approx \alpha_{11},]\underline{\mathsf{int} \approx \alpha_{11}}, \underline{\alpha_5 \approx \alpha_{14}}, \alpha_1 \approx \mathsf{list}(\alpha_{14}) \to \mathsf{int}\}$

$[\alpha_2 := \mathsf{list}(\alpha_{14}), \alpha_{11} := \mathsf{int}, \alpha_5 := \alpha_{14}]$

$$\{\ [\alpha_1 \approx \mathsf{list}(\alpha_{14}) \to \mathsf{int},]\underline{\mathsf{list}(\alpha_{14}) \approx \mathsf{list}(\alpha_{13})}, \alpha_1 \approx \mathsf{list}(\alpha_{14}) \to \mathsf{int}\}$$

$$\{\alpha_{14} \approx \alpha_{13}, \alpha_1 \approx \mathsf{list}(\alpha_{14}) \to \mathsf{int}\}$$

$[\alpha_{13} := \alpha_{14}]$

$$\{\alpha_1 \approx \mathsf{list}(\alpha_{14}) \to \mathsf{int}\}$$

Ergo, a mgu σ exists such that $\sigma(\alpha_1) = \mathsf{list}(\alpha_{14}) \to \mathsf{int}$. This means that len has type $\mathsf{list}(\alpha_{14}) \to \mathsf{int}$, which we can rename to $\mathsf{list}(\alpha) \to \mathsf{int}$

3.2 Formal definition

To properly define what an instance is, we recall the definition of variables in a term translated to type expressions:

$$\begin{array}{ll} \mathsf{Var}(x) & = \{x\} \\ \mathsf{Var}(\tau_1 \to \tau_2) & = \mathsf{Var}(\tau_1) \cup \mathsf{Var}(\tau_2) \\ \mathsf{Var}(c(\tau_1, \cdots, \tau_n) & = \mathsf{Var}(\tau_1) \cup \cdots \cup \mathsf{Var}(\tau_n) \end{array}$$

Then we can define

Definition 3.1 A type expression τ' is a (fresh) instance of a type expression τ if

$$\tau' = \tau[\alpha_1 := \beta_1, \cdots, \alpha_n := \beta_n]$$

where

$$\mathsf{Var}(\tau) = \{\alpha_1, \cdots, \alpha_n\} \text{ (and } \beta_1, \cdots, \beta_n \text{ are fresh)}$$

Definition 3.2 The labeled version of

$$\mathsf{letrec}\ x = M\ ;;$$

is

$$\mathsf{letrec}\ x : \alpha = \mathcal{L}(x : \alpha, M)\ ;;$$

where α is fresh and

$$\mathcal{L}(A,M) = \begin{cases} \overline{x} : A(x) \\ \quad M = x, A(x) \neq \perp \\ \underline{x} : \alpha \\ \quad M = x, A(x) = \perp, \alpha \text{ fresh} \\ (\lambda x : \alpha_1 . \mathcal{L}(A, x : \alpha_1, M')) : \alpha_2 \\ \quad M = \lambda x.M', \alpha_1, \alpha_2 \text{ fresh} \\ (\mathcal{L}(A, M_1), \mathcal{L}(A, M_2)) : \alpha \\ \quad M = M_1, M_2, \alpha \text{ fresh} \\ c(\mathcal{L}(A, M_1), \cdots, \mathcal{L}(A, M_2)) : \alpha \\ \quad M = c(M_1, \cdots, M_n)\alpha \text{ fresh} \\ (\mathsf{let}\ x : \alpha_1 = \mathcal{L}(A, M_1)\ \mathsf{in}\ \mathcal{L}(A, x : \alpha_1, M_2)) : \alpha_2 \\ \quad M = \mathsf{let}\ x = M_1\ \mathsf{in}\ M_2, \alpha_1, \alpha_2 \text{ fresh} \\ (\mathsf{match}\, \mathcal{L}(A, M_0)\ \mathsf{with} \quad \mid \mathcal{L}(A, A_1, p_1)\ \mathsf{when}\ \mathcal{L}(A, A_1, c_1) \rightarrow \mathcal{L}(A, A_1, M_1) \\ \qquad \vdots \\ \qquad \mid \mathcal{L}(A, A_n, p_n)\ \mathsf{when}\ \mathcal{L}(A, A_n, c_n) \rightarrow \mathcal{L}(A, A_n, M_n)) : \alpha \\ \quad M = \mathsf{match}\, M_0\ \mathsf{with} \mid p_1\ \mathsf{when}\ c_1 \rightarrow M_1 \cdots \mid p_n\ \mathsf{when}\ c_n \rightarrow M_n, \\ \quad \alpha \text{ fresh}, A_i = \mathsf{alloc}(p_i) \end{cases}$$

and

$$\mathsf{alloc}(p) = \begin{cases} x : \alpha & p = x, \alpha \text{ fresh} \\ \mathsf{alloc}(p_1), \cdots, \mathsf{alloc}(p_n) & p = c(p_1, \cdots, p_n) \end{cases}$$

Definition 3.3 Given and environment (C, F) a labeled recursive definition

$$\mathsf{letrec}\ x : \alpha_1 = M : \alpha_2\ ;;$$

The generated equations are

$$\alpha_1 \approx \alpha_2, \mathsf{eqs}(M : \alpha_2)$$

where

$$
\mathsf{eqs}(M:\alpha) = \begin{cases}
\epsilon \\
\quad M = \overline{x} \\
\alpha \approx \tau \\
\quad M = \underline{x}, \tau \text{ fresh instance of } F(x) \neq \perp \\
\alpha \approx \alpha_1 \to \alpha_2, \mathsf{eqs}(M' : \alpha_2) \\
\quad M = \lambda x : \alpha_1 . M' : \alpha_2 \\
\alpha_1 = \alpha_2 \to \alpha, \mathsf{eqs}(M_1 : \alpha_1), \mathsf{eqs}(M_2 : \alpha_2) \\
\quad M = (M_1 : \alpha_1)(M_2 : \alpha_2) \\
\alpha \approx \tau \\
\quad M = c, T(c) = 0, \tau \text{ fresh instance of } C(c) \\
\alpha \approx \tau, \mathsf{eqs}(M_1 : \tau_1), \cdots, \mathsf{eqs}(M_n : \tau_2) \\
\quad M = c(M_1, \cdots, M_n), T(c) = n > 0, \\
\quad (\tau_1 * \cdots * \tau_n \to \tau) \text{ fresh instance of } \mathsf{inst}(C(c)) \\
\alpha \approx \alpha_2, \alpha_0 \approx \alpha_1, \mathsf{eqs}(M_1 : \alpha_1), \mathsf{eqs}(M_2 : \alpha_2) \\
\quad M = \mathsf{let}\ x : \alpha_0 = M_1 : \alpha_1\ \mathsf{in}\ M_2 : \alpha_2 \\
\alpha \approx \alpha_1, \cdots, \alpha \approx \alpha_n, \mathsf{eqs}(M_1 : \alpha_1), \cdots, \mathsf{eqs}(M_n : \alpha_n), \\
\beta \approx \beta_1, \cdots, \beta \approx \beta_n, \mathsf{eqs}(p_1 : \beta_1), \cdots, \mathsf{eqs}(p_n : \beta_n), \\
\gamma_1 \approx \mathsf{bool}, \cdots, \gamma_n \approx \mathsf{bool}, \mathsf{eqs}(c_1 : \gamma_1), \cdots, \mathsf{eqs}(c_n : \gamma_n) \\
\quad M = \mathsf{match}\ M_0 : \beta\ \mathsf{with}\ \begin{array}{l} \mid p_1 : \beta_1\ \mathsf{when}\ c_1 : \gamma_1 \to M_1 : \alpha_1 \\ \vdots \\ \mid p_n : \beta_n\ \mathsf{when}\ c_n : \gamma_n \to M_n : \alpha_n \end{array}
\end{cases}
$$

4 Exercises

Exercise 4.1 Implement unification on type expressions. The algorithm works equally well if E is a list rather than a set, so you can represent a unification problem as a list of pairs of type expressions. For example, the problem

$$\alpha_1 \approx \alpha_2 \to \alpha_3, \alpha_3 \approx \alpha_4 \to \alpha_5, \alpha_2 \approx \alpha_5$$

translates to

```
[(TVar"a1",TCons("->",[TVar"a2";TVar "a3"]));
 (TVar"a3",TCons("->",[TVar"a4";TVar "a5"]));
 (TVar"a2",TVar"a5")]
```

The following modules are provided:

Aux A module with auxiliary functions.

Map An abstract data type that can map a finite set of keys to values.

Types A module that contains a data type for type expression. We use maps as finite subtitutions. Applying substitution to a term and the composition of two substitutions have already been implemented.

Exercise 4.2 Let $\tau_1 = f(\alpha, f(\beta, f(\alpha, beta)))$ and $\tau_2 = \alpha \to \beta \to \alpha$. Compute $\tau_1\sigma$ and $\tau_2\sigma$ for

a $\sigma = [\alpha := \beta]$.
b $\sigma = [\alpha := f(\alpha, \beta), \beta := f(\beta, \beta), \gamma := \alpha]$
c $\sigma = [\alpha := \mathsf{int}, \beta := \mathsf{list}(\mathsf{bool}), \gamma := \alpha \rightarrow \beta]$

Exercise 4.3 Given
$\sigma_1 = [\alpha := \alpha \rightarrow \alpha]$
$\sigma_2 = [\alpha := \beta, \beta := \alpha]$
$\sigma_3 = [\beta := \alpha \rightarrow \gamma]$
Compute
a $\sigma_1\sigma_2$
b $\sigma_2\sigma_1$
c $\sigma_1\sigma_3$
d $\sigma_3\sigma_1$
e $\sigma_2\sigma_3$
f $\sigma_3\sigma_2$

Exercise 4.4 Compute a mgu (if one exists) for the following sets of equations:
a $\{f(g(\alpha, \beta), \alpha, \beta) \approx f(\gamma, g(\beta, \beta), \beta)\}$
b $\{g(h(\alpha), g(\alpha, \beta)) \approx g(\gamma, g(g(\alpha, \alpha), \gamma))\}$
c $\{f(\alpha, g(\alpha, \beta), h(\beta)) \approx f(g(\gamma, \gamma), \alpha, \alpha)\}$.

Exercise 4.5 Below, we define the function int_enum, which takes an initial value n and returns a function, which returns on $n + i - 1$ on the ith call. (That is, it returns $n, n + 1, n + 2, \cdots$.)

```
let int_enum first =
  let n = ref first in
  fun () -> let i = !n in n := i+1 ;i
;;
```

Write a function var_enum that takes a base string `s` and returns a function that will return `s0`, `s1`, `s2`, etc. For example, with that function we should be able to get the following results.

```
# let fresh =  var_enum "alpha";;
val fresh : unit -> string = <fun>
# fresh();;
- : string = "alpha0"
# fresh();;
- : string = "alpha1"
# fresh();;
- : string = "alpha2"
```

Other useful functions: string concatenation (^), and int $\leftrightarrow$ string conversion: string_of_int and int_of_string .

Exercise 4.6 Write a function variant that given a generator for fresh variables and a type expression returns a fresh variant of the type expression. For example

```
# let t = TCons("list",[TVar "a2"]);;
val t : Types.type_expr = TCons ("list", [TVar "a2"])
# let fresh = var_enum "a";;
val fresh : unit -> string = <fun>
# variant fresh t;;
- : Types.type_expr = TCons ("list", [TVar "a0"])
# variant fresh t;;
- : Types.type_expr = TCons ("list", [TVar "a1"])
# variant fresh t;;
- : Types.type_expr = TCons ("list", [TVar "a2"])
# variant fresh t;;
- : Types.type_expr = TCons ("list", [TVar "a3"])
```

Note that when we ask for the third variant, we actually get the original type back. This is correct behavior because the variables used in universal types use a separate name space from the variables used for labeling.

Exercise 4.7 Manually infer the types of Compute a mgu (if one exists) for the following sets of equations:
a $\lambda x.x$
b $\lambda x\, y.x$
c $\lambda x\, y\, z.(x\, z)(y\, z)$

Exercise 4.8 Manually infer the type of

$$\mathsf{letrec}\ reva\ ;;\ \mathsf{fun}\ a \to \mathsf{fun}\ xs \to \mathsf{match}\ xs\ \mathsf{with} |\ [] \to a \mid x :: xs \to reva(x :: a)xs$$