Typing Rules for Sequential MPL

March 19, 2017

1 Type Inferencing

1.1 Introduction

This chapter deals with the type inference of various MPL constructs, namely functions, processes, terms and process commands when they are not type annotated and type checking when they are. Type inference process involves two main steps.

- **Generating Type Equations** - Type Equations represent the constraint relationship between the different parts of the construct being type inferred. $\exists$ used in the typing rules here should not be viewed existential quantification from the predicate logic rather it is used a notation for type binder.

  Using the typing rules, a series of equations are generated going from botton to top in the type derivation.This is also called the proof search direction. The equation structure is not linear rather it is hierarchical capturing the shape of the type inference tree. Thus, the placement of an equation inside a type equation depends on thes the location within a type inference tree that resulted in that equation within the type equation. Advantages of this approach are faster solution of type equations and better localisation of type errors.

- **Solving Type Equations** - The type equations hence forth generated are solved in order to get the most general type of a given construct.

In this chapter, the type equations generation for the Sequential MPL (terms and functions) is dicussed first followed by that of the Concurrent MPL (process commands and processes). This is followed by an alogrithm to solve the type equations to get the most general type.

2 Generating Type Equations for sequential terms

2.1 var term

In order to infer a variable,the variable should be looked up in the context. If it is present in the context, then the type variable corresponding to the var term should be equated with the variable type from the context.

Typing rule for var

$$\frac{}{x : P, \Gamma \vdash x : Q \quad \left\langle P = Q \right\rangle} \text{ id}$$

2.2 default term

default term is used as the catch all branch while using guards with the switch statement. The type of the default term is always Bool.

Typing rule for default

$$\frac{}{\Gamma \vdash default : T \quad \left\langle T = Bool \right\rangle} \text{ default}$$

2.3 constant term

constants can be one of the built in constant types, like int, double and char.

Typing rule for constants

$$\frac{\Gamma \vdash n : Int}{\Gamma \vdash \ n : T \quad \left\langle T = Int \right\rangle} \text{ int}$$

$$\frac{\Gamma \vdash n : Double}{\Gamma \vdash \ n : T \quad \left\langle T = Double \right\rangle} \text{ double}$$

$$\frac{\Gamma \vdash n : Char}{\Gamma \vdash \ n : T \quad \left\langle T = Char \right\rangle} \text{ char}$$

2.4 case and cons term

cons term creates a data with the constructors of that data type.

case term branches on the different constructors of the data type. Every branch has a constructor of the data type with some arguments on the left and a term on the right of the arrow.

Let us define a data type D. C_1 ... C_m are the different constructors and A_1, ..., A_k are the union of type variables used in the different constructors of the data type D. For C_1 ,F_{11}, ..., F_{1a} repesent the input type of the constructor.

$$\begin{aligned} \forall A_1,\ \ldots,\ A_k.\ data\ D\ \rightarrow\ A = \ & C_1 : F_{11},\ \ldots,\ F_{1a} \rightarrow A \\ & \vdots \qquad\quad \vdots \qquad\quad \vdots \\ & C_m : F_{m1},\ \ldots,\ F_{mn} \rightarrow A \end{aligned}$$

The different constructs above are α-renamed with fresh variable A'_1, ..., A'_k to avoid any conflict between the variable names used in the the data construct with that used in the equation. We will use the renaming technique while type inferring other MPL constructs as well because of the same reason. The renamed constructs have been represented by adding a superscript $'$ to the name of the construct.

$$\begin{aligned} D' \quad &= (\Lambda A_1, \ldots, A_k.\ D)\ A'_1, \ldots, A'_k \\ F'_{xy} &= (\Lambda A_1, \ldots, A_k.\ F_{xy})\ A'_1, \ldots, A'_k \quad \{1 \leq x \leq m,\ 1 \leq y \leq p \mid p \in \{a, b, \ldots, n\}\} \end{aligned}$$

The output type of the cons term is the data type of which it is a constructor of. The function type of the constructor is looked up from the symbol table.The different arguments of the constructor are assigned their corresponding types from the input types of the afore mentioned function type. E_1, ..., E_n are the type equations generated for the terms t_1, ..., t_n respectively.

The type of a case term is the type of the terms on the right side of the branches. A well typed case term will have same type for all the terms on the right side of the branches. The types of the input arguments for a constructor on the left hand side of a branch is obtained by looking up the function type for that constructor from the symbol table.

Typing rule for cons and case

$$\frac{\Gamma \vdash t_1 : T_1 \quad \langle E_1 \rangle \qquad \ldots \qquad \Gamma \vdash t_j : T_j \quad \langle E_j \rangle}{\Gamma \vdash \mathsf{cons}\ (C_i, [t_1, \ldots, t_j]) : T \quad \left\langle \exists \begin{array}{ll} T_1, \ldots, T_j, & T = D' \\ A_1, \ldots, A_n. & T_1 = F'_{i1},\ \ldots,\ T_j = F'_{ij} \\ & E_1, E_2, \ldots, E_j \end{array} \right\rangle}\ \text{cons}$$

$$\frac{\Gamma \vdash t : T_0\ \langle E_0 \rangle \qquad \begin{array}{lll} \Gamma, x_{m1} : T_{m1}, \ldots, x_{mn} : T_{mn} & \vdash & t_m : T_m \quad \langle E_m \rangle \\ \vdots & & \\ \Gamma, x_{11} : T_{11}, \ldots, x_{1a} : T_{1a} & \vdash & t_1 : T_1 \quad \langle E_1 \rangle \end{array}}{\Gamma \vdash \begin{array}{l} \mathsf{case}\ t \\ \mathsf{of} \end{array} \left| \begin{array}{lll} C_1\ x_{11}, \ldots, x_{1a} & \rightarrow & t_1 \\ \vdots & \vdots & \vdots \\ C_m\ x_{m1}, \ldots, x_{mn} & \rightarrow & t_n \end{array} \right. : T \quad \left\langle \exists \begin{array}{ll} T_0, T_1, \ldots, T_m, & T_0 = D_N, T_1 = T, \ldots, T_m = T \\ T_{11}, \ldots, T_{1a}, & T_{11} = F'_{11}, \ldots, T_{11} = F'_{1a} \\ \vdots & \vdots \\ T_{m1}, \ldots, T_{mn}, & T_{m1} = F'_{m1}, \ldots, T_{mn} = F'_{mn} \\ A'_1, \ldots, A'_k & E_1, E_2, \ldots, E_m \end{array} \right\rangle}\ \text{case}$$

2.5 fold term

fold is used to implement higher order primitive recursive functions which gurantees termination. Like case, fold also has branches corresponding to every constructor of the data type which is being folder. Every branch has some argument on the left side of the arrow and a term on the right side.

For all the constructors of a data type, their fold functions are inserted in the symbol table. We will use the data defintion D and the renamed constructs from the previous section on case and cons.

The type of a fold term is the type of the terms on the right side of the branches. A well typed fold term will have same type for all the terms on the right side of the branches. The types of the input arguments for a given constructor on the left hand side of a branch is obtained by looking up the fold function type for that constructor from the symbol table.

Typing rule for fold

$$\frac{\begin{array}{ll} & x_{m1}:T_{m1},\ldots,x_{mn}:T_{mn} \vdash t_m:T_m \quad \langle E_m\rangle \\ & \qquad\qquad\qquad\qquad\qquad\qquad \vdots \\ t:T_0\ \langle E_0\rangle & x_{11}:T_{11},\ldots,x_{1a}:T_{1a} \vdash t_1:T_1 \quad \langle E_1\rangle \end{array}}{\Gamma \vdash \begin{array}{l}\mathsf{fold}\ t \\ \mathsf{of} \\ \end{array} \left|\begin{array}{lllll} C_1 & : & x_{11},\ldots,x_{1a} & \to & t_1 \\ \vdots & & & \vdots & \vdots \\ C_m & : & x_{m1},\ldots,x_{mn} & \to & t_n \end{array}\right. : T}\ \text{fold}$$

$$\left\langle \exists \begin{array}{ll} T_0,T_1,\ldots,T_m, & T_0 = D' \\ T_{11},\ldots,T_{1a}, & T_1 = T,\ldots,T_m = T \\ \vdots & T_{11} = F'_{11},\ldots,T_{11} = F'_{1a} \\ T_{m1},\ldots,T_{mn}, & \vdots \\ A'_1,\ldots,A'_k & T_{m1} = F'_{m1},\ldots,T_{mn} = F'_{mn} \\ & E_1,E_2,\ldots,E_m \end{array} \right\rangle$$

2.6 record and destructor term

record term forms a record for a codata type. dest destructs a record of a codata type with a particular destuctor of that codata type.It obtains the term corresponding to that destructor from the record.

Let us define a codata type C. D_1, ..., D_m are the different destructors of the codata type C and A_1, ..., A_k are the union of type variables used in these destructors. For destructor D_1, F_{11}, ..., F_{1a} repesent the types of its input argument.

$$\begin{array}{lll} \forall A_1,\ \ldots,\ A_k.\ codata\ A\ \to\ C = & D_1 : F_{11},\ \ldots,\ F_{1a},\ A \to O_1 \\ & \vdots \qquad\quad \vdots \qquad\qquad \vdots \\ & D_m : F_{m1},\ \ldots,\ F_{mn},\ A \to O_n \end{array}$$

Renaming the different constructs in the codata C with fresh variables A'_1, ..., A'_k gives the following constructs.

$$\begin{array}{lll} C' & = (\Lambda A_1,\ldots,A_k.\ C)\ A'_1,\ldots,A'_k & \\ F'_{xy} & = (\Lambda A_1,\ldots,A_k.\ F_{xy})\ A'_1,\ldots,A'_k & \{1 \le x \le m, 1 \le y \le p \mid p \in \{a,b,\ldots,n\}\} \\ O'_i & = (\Lambda A_1,\ldots,A_k.\ O_i)\ A'_1,\ldots,A'_k & \{1 \le i \le m\} \end{array}$$

The type of a record term for a codata type is that codata type. The types of the different destructor arguments and the term corresponding to a destructor branch of the record is obtained by looking up the function type corresponding to that destructor from the symbol table. The input types of the function type give the type of the destructor arguments and the output type gives the type of

the term corresponding to the destructor branch.

The type of a destructor term is the type of term corresponding to that destructor branch in the given record. This is obtained from the function type (output type of the function type) of the destructor which can be looked up from the symbol table. The types of the destructor arguments are obtained from the corresponding input type of the afore mentioned function type.

Typing rule for record and dest

$$\frac{\begin{array}{lll} & x_{11}:T_{11},\ldots,x_{1a}:T_{1a} \vdash t_1:T_1 \quad \langle E_1\rangle \\ & \vdots \\ t:T_0 \quad \langle E_0\rangle & x_{m1}:T_{m1},\ldots,x_{mn}:T_{mn} \vdash t_m:T_m \quad \langle E_m\rangle \end{array}}{\Gamma \vdash \begin{array}{l}\mathsf{rec} \\ \mathsf{of}\end{array} \left|\begin{array}{lll} D_1\ x_{11},\ldots,x_{1a} & \to & t_1 \\ \vdots & \vdots & \vdots \\ D_m\ x_{m1},\ldots,x_{mn} & \to & t_m \end{array}\right. : T} \text{ rec}$$

$$\left\langle \exists \begin{array}{l} T_0,T_1,\ldots,T_m, \\ T_{11},\ldots,T_{1a}, \\ \quad\vdots \\ T_{m1},\ldots,T_{mn}, \\ A'_1,\ldots,A'_k \end{array} . \begin{array}{l} T=C_N \\ T_1=O_{1,N},\ldots,T_m=O_{m,N} \\ T_{11}=F_{11,N},\ldots,T_{1a}=F_{1a,N} \\ \quad\vdots \\ T_{m1}=F_{m1,N},\ldots,T_{mn}=F_{mn,N} \\ E_1,E_2,\ldots,E_m \end{array} \right\rangle$$

$$\frac{\begin{array}{ll} \Gamma \vdash t_{i1}:T_1 \quad \langle E_{i1}\rangle & \\ \vdots & \\ \Gamma \vdash t_{ij}:T_j \quad \langle E_{ij}\rangle & \Gamma, x_{i1}:S_1,\ldots,x_{ij}:S_j \vdash t_i:T_0 \quad \langle E_0\rangle \end{array}}{\Gamma \vdash \mathsf{dest}\ (D_i,[t_{i1},\ldots,t_{ij}]) \left(\begin{array}{l}\mathsf{rec} \\ \mathsf{of}\end{array} \left|\begin{array}{lll} \vdots & \vdots & \vdots \\ D_i\ x_{i1},\ldots,x_{ij} & \to & t_i \\ \vdots & \vdots & \vdots \end{array}\right.\right) : T} \text{ dest}$$

$$\left\langle \exists \begin{array}{l} T_0,T_1,\ldots,T_j, \\ A'_1,\ldots,A'_n \end{array} . \begin{array}{l} T=T_0, \\ T_0=O'_i,T_1=S_1,\ldots,T_j=S_j \\ S_1=F'_{i1},\ldots,S_j=F'_{ij}, \\ E_0,E_1,\ldots,E_j \end{array} \right\rangle$$

2.7 prod term

prod term is used to reprsent tuples in MPL.The output type of a tuple is the product of the types of the individual elements.

Typing rule for prod

$$\frac{\Gamma \vdash x_1 : T_1 \;\; \langle E_1 \rangle \qquad \ldots \qquad \Gamma \vdash x_n : T_n \;\; \langle E_n \rangle}{\Gamma \vdash (x_1, \ldots, x_n) : T \;\; \left\langle \exists\, T_1, \ldots, T_n \;.\; \begin{array}{l} T = (T_1, \ldots, T_n) \\ , E_1, \ldots, E_n \end{array} \right\rangle} \text{ prod}$$

2.8 function call

Function call is used to call an already defined function in the program. The type of the called function is looked up from the symbol table. In the below representation, function f's type is comprised of m input types I_1, ... I_m and an output type O.

$$f : \forall A_1, \;\; \ldots, \;\; A_k.\; S_1, \ldots, S_m \rightarrow O$$

As in the previous term case, the output and input types are α renamed.

$$S'_i = (\Lambda A_1, \ldots, A_k.\; I_i)\; A'_1, \ldots, A'_k \quad \{\; 1 \;\leq\; i \;\leq m\}$$
$$O' = (\Lambda A_1, \ldots, A_k.\; O)\; A'_1, \ldots, A'_k$$

The type of a function call is the type of the output type of the function being called. To type infer the function arguments, the arguments are type inferred in the original context Γ and these types are equated with their corresponding input types of the function type f.

Typing rule for function call

$$\frac{\Gamma \vdash x_1 : X_1 \;\; \langle E_1 \rangle \qquad \ldots \qquad \Gamma \vdash x_n : X_n \;\; \langle E_n \rangle}{\Gamma \vdash \mathsf{f}\;(\mathsf{x_1}, \ldots, \mathsf{x_m}) : T \;\; \left\langle \exists \;\; \begin{array}{l} A'_1, \ldots, A'_k, \\ T_1, \ldots, T_m \end{array} \;.\; \begin{array}{l} T = O', \\ T_1 = S'_1, \ldots, T_m = S'_m, \\ E_1, \ldots, E_m \end{array} \right\rangle} \text{ call}$$

2.9 if term

if is a term with three arguments, a boolean expression and two statements. Depending on the boolean value being True or False, first or the second statement is executed. The type of the if statement is the type of these statements. A correctly typed if term will have these two statements of the same type.

Typing rule for if

$$\frac{\Gamma \vdash t_1 : T_1 \;\; \langle E_1 \rangle \qquad \Gamma \vdash t_2 : T_2 \;\; \langle E_2 \rangle \qquad \Gamma \vdash t_3 : T_3 \;\; \langle E_3 \rangle}{\Gamma \vdash \mathsf{if}\ \mathsf{t_1}\ \mathsf{then}\ \mathsf{t_2}\ \mathsf{else}\ \mathsf{t_3} : T \quad \left\langle \exists\, T_1, T_2, T_3 \;.\; \begin{array}{l} T_1 = Bool, \\ T_2 = T, T_3 = T, \\ E_1, E_2, E_3 \end{array} \right\rangle} \text{ if}$$

2.10 Let term

let statment allows local defintions to be used in a term. let also allows the programmer define local functions. The local functions are type inferred and added to the symbol table. The scope of these functions is just the let term. The representation below assumes that these local functions have already been taken care of and all that remains in the where part are local variable/pattern declarations.

The type the let statement is the type of the term **t**. The term t is type inferred in the context Γ extended with the variables defined in the where part of the let term.

Typing rule for let

$$\frac{\begin{array}{l} \Gamma \vdash t_m : T_m \;\; \langle E_m \rangle \\ \quad \vdots \\ \Gamma \vdash t_1 : T_1 \;\; \langle E_1 \rangle \end{array} \qquad \Gamma, x_1 : S_1, \ldots x_m : S_m \vdash t : T_0 \;\; \langle E_0 \rangle}{\Gamma \vdash \begin{array}{l} \mathsf{let}\ t \\ \mathsf{where} \\ \quad \left| \begin{array}{lcl} x_1 & = & t_1 \\ \vdots & \vdots & \vdots \\ x_m & = & t_m \end{array} \right. \end{array} : T \quad \left\langle \exists \begin{array}{l} T_0, \ldots, T_m, \\ S_1, \ldots, S_m \end{array} \;.\; \begin{array}{l} T = T_0, \\ S_1 = T_1, \ldots, S_m = T_m, \\ E_0, E_1, \ldots, E_m \end{array} \right\rangle} \text{ let}$$

3 Generating Type Equations for Patterns

Patterns are MPL constructs used in defining functions and processes. Function, processes and the let term in MPL have array of pairs of patterns and terms, i.e corresponding to a list of patterns in every row in constructs mentioned above there is a term or a process command. Before we go

ahead and type infer functions,processes or term, lets write down the typing rules for patterns first. The different types of patterns for which we will see the typing rules are as follows.

- Variable Pattern
- Int Pattern
- Double Pattern
- Char Pattern
- Constructor Pattern
- Product Pattern

3.1 Typing rules for patterns

Since there may be a list of patterns in the constructs where they are used, we will type infer the patterns as a list. Every pattern in the list enhances this context and generates some equations. The final context is used to type infer the term on the right side of the patterns. This is because the final context has all the variables in scope from the patterns on the left.

3.1.1 variable pattern

The below typing rule is for a list of patterns where variable pattern x is the first pattern followed by the rest of the patterns r.

Typing rule for var pattern

$$\frac{\Gamma, x : P \vdash \phi \quad \left\langle \langle \Gamma_1 = x : P, \Gamma \rangle, \langle \phi \rangle \right\rangle \qquad \Gamma_1 \vdash r : R \quad \left\langle \langle \Gamma_f \rangle, \langle E \rangle \right\rangle}{\Gamma \vdash \mathsf{varPatt}\ \ \mathsf{x}\ .\ r\ \ : T \quad \left\langle \langle \Gamma_f \rangle, \left\langle \exists R\ .\ E \right\rangle \right\rangle}\ \text{varPatt}$$

3.1.2 int pattern

Typing rule for int pattern

$$\frac{n : Int \qquad \Gamma \vdash r : R \quad \left\langle \langle \Gamma_f \rangle, \langle E \rangle \right\rangle}{\Gamma \vdash \mathsf{intPatt}\ \ \mathsf{n}\ .\ r\ \ : T \quad \left\langle \langle \Gamma_f \rangle, \left\langle \exists R\ .\ T = Int,\quad E \right\rangle \right\rangle}\ \text{intPatt}$$

3.1.3 double pattern

Typing rule for double pattern

$$\frac{n : Double \qquad \Gamma \vdash r : R \quad \left\langle \langle \Gamma_f \rangle, \langle E \rangle \right\rangle}{\Gamma \vdash \mathsf{doublePatt}\ \ \mathsf{n}\ .\ r\ \ : T \quad \left\langle \langle \Gamma_f \rangle, \left\langle \exists R\ .\ T = Double,\quad E \right\rangle \right\rangle}\ \text{doublePatt}$$

3.1.4 char pattern

Typing rule for char pattern

$$\frac{n : Char \qquad \Gamma \vdash r : R \quad \left\langle \langle \Gamma_f \rangle, \langle E \rangle \right\rangle}{\Gamma \vdash \mathsf{charPatt}\ \ \mathsf{n}\ .\ r\ \ : T \quad \left\langle \langle \Gamma_f \rangle, \left\langle \exists R\ .\ T = Char,\quad E \right\rangle \right\rangle}\ \text{charPatt}$$

3.1.5 constructor pattern

Constructor patterns are made up of constructors of a data type. Let us consider this data type to be D. $C_1 \ldots C_m$ are the different constructors and $A_1, \ldots, A_k$ are the union of type variables used in the different constructors of the data type D. For C_1 ,$F_{11}, \ldots, F_{1a}$ repesent the input type of the constructor.

$$\begin{aligned} \forall A_1,\ \ldots,\ A_k.\ data\ D\ \rightarrow\ A = \ & C_1 : F_{11},\ \ldots,\ F_{1a} \rightarrow A \\ & \vdots \qquad\quad \vdots \qquad\quad \vdots \\ & C_m : F_{m1},\ \ldots,\ F_{mn} \rightarrow A \end{aligned}$$

The different constructs above are α-renamed with fresh variable $A'_1, \ldots, A'_k$ to avoid any conflict between the variable names used in the the data construct with that used in the equation. The renamed constructs have been represented by adding a superscript $\prime$ to the name of the construct.

$$\begin{aligned} D' \quad &= (\Lambda A_1, \ldots, A_k.\ D)\ A'_1, \ldots, A'_k \\ F'_{xy} &= (\Lambda A_1, \ldots, A_k.\ F_{xy})\ A'_1, \ldots, A'_k \quad \{1 \leq x \leq m,\ 1 \leq y \leq p \mid p \in \{a, b, \ldots, n\}\} \end{aligned}$$

combEqns is a special function used in the typing rule below. This function takes list of type equations as input. The type equation can be of two types.

- Type equations with exisential and universal variables.
- Type equations without exisential and universal variables.

combEqns function takes a list of type equations and concatenates their universal variables, existential variables and type equations. In the case the list contains a mixture of the two kinds of type equations mentioned above, the resultant type equation will be one type type equation with existential and universal variables. In case all the equations are without variables of either kind, the output list is same as the input list.

Typing rule for constructor pattern

$$\frac{\begin{array}{lll} \Gamma_{n-1} \vdash t_n : T_n & \left\langle \langle \Gamma_n \rangle, \langle E_n \rangle \right\rangle & \\ \vdots & \vdots & \\ \Gamma \vdash t_1 : T_1 & \left\langle \langle \Gamma_1 \rangle, \langle E_1 \rangle \right\rangle & \Gamma_n \vdash r : R \quad \left\langle \langle \Gamma_r \rangle, \langle E_r \rangle \right\rangle \end{array}}{\begin{array}{l} \Gamma \vdash \mathsf{consPatt}\ (\mathsf{C_i}, [\mathsf{t_1}, \ldots, \mathsf{t_j}]\ )\ .\ r\ : T \\ \left\langle \langle \Gamma_r \rangle, \left\langle \exists\ \begin{array}{ll} R, & T = D', \\ A'_1, \ldots A'_k & T_1 = F'_{i1}, \ldots, T_j = F'_{ij}, \\ T_1, \ldots, T_n\ . & E_1, E_2, \ldots, E_n, E_r \end{array} \right\rangle \right\rangle \end{array}}\ \text{consPatt}$$

3.1.6 product pattern

Typing rule for product pattern

$$\frac{\begin{array}{llll}\Gamma_{n-1} \vdash t_n : T_n & \left\langle \langle \Gamma_n \rangle, \langle E_n \rangle \right\rangle & & \\ \vdots & \vdots & & \\ \Gamma \vdash t_1 : T_1 & \left\langle \langle \Gamma_1 \rangle, \langle E_1 \rangle \right\rangle & \Gamma_n \vdash r : R & \left\langle \langle \Gamma_r \rangle, \langle E_r \rangle \right\rangle \end{array}}{\begin{array}{l}\Gamma \vdash (\mathsf{t_1}, \ldots, \mathsf{t_n}) \,.\, r \;\; : T \\ \left\langle \langle \Gamma_f \rangle, \left\langle \exists R, T_1, \ldots, T_n. \begin{array}{l} T = (T_1, T_2, \ldots, T_n), \\ E_1, E_2, \ldots, E_n, E_r \end{array} \right\rangle \right\rangle \end{array}} \text{ prodPatt}$$

4 Generating Type Equations for Function Definitions

4.1 Function definitions without an annotated type

These are the function definitions for which the programmer has not annotated the expected type of the function. Once the function defintion is type inferred, the function name is inserted in the symbol table with this type.

Typing rule for function defns with variable patterns

$$\frac{\Gamma \vdash p_1, \ldots, p_m : P \;\; \langle \Gamma_p, E_p \rangle \qquad \Gamma_p \vdash t : O \;\; \langle E \rangle}{\begin{array}{l}\Gamma \vdash \begin{array}{l}\mathsf{fun\ f} \\ \quad \mid \; p_1, \ldots, p_m \;\; \rightarrow \;\; t \end{array} \quad : T \\ \left\langle \exists \; S_1, \ldots, S_m \;.\; \begin{array}{l} T = (S_1, \ldots, S_m) \;\rightarrow\; O, \\ E \end{array} \right\rangle \end{array}} \text{ fdefn}_1$$

Suppose the patterns on the left hand side in the function defintion are constructors of a data

type D. $C_1 \ldots C_m$ are the different constructors of the data type D and $A_1, \ldots, A_k$ are the union of type variables used in the different constructors of the data type.

$$\begin{array}{lll} \forall A_1,\ \ldots,\ A_k.\ data\ D\ \rightarrow\ A = & C_1 : F_{11},\ \ldots,\ F_{1a} \rightarrow A \\ & \vdots \quad\quad \vdots \quad\quad \vdots \\ & C_m : F_{m1},\ \ldots,\ F_{mn} \rightarrow A \end{array}$$

α renaming the different constructs in the data D with fresh variables $A'_1, \ldots, A'_k$ gives the following constructs.

$$\begin{array}{ll} D' & = (\Lambda A_1, \ldots, A_k.\ D)\ A'_1, \ldots, A'_k \\ F'_{xy} & = (\Lambda A_1, \ldots, A_k.\ F_{xy})\ A'_1, \ldots, A'_k \quad \{1 \leq x \leq m,\ 1 \leq y \leq p \mid p \in \{a, b, \ldots, n\}\} \end{array}$$

Typing rule for function defns with constructor patterns

$$\frac{\begin{array}{l} \Gamma, x_{11} : S_{11},\ \ldots,\ x_{1a} : S_{1a} \quad \vdash\ t_1 : T_1 \quad \langle E_1 \rangle \\ \quad\quad\quad\quad \vdots \\ \Gamma, x_{m1} : S_{m1},\ \ldots,\ x_{mn} : S_{mn} \quad \vdash\ t_m : T_m \quad \langle E_m \rangle \end{array}}{\Gamma \vdash \begin{array}{l} \mathsf{fun\ f} \\ \left|\begin{array}{lll} C_1\ x_{11}, \ldots, x_{1a} & \rightarrow & t_1 \\ \vdots & \vdots & \vdots \\ C_m\ x_{m1}, \ldots, x_{mn} & \rightarrow & t_m \end{array}\right. \end{array} : T}\ \text{fdefn}_2$$

$$\left\langle \exists \begin{array}{ll} S_{11}, \ldots, S_{1a} & \\ \vdots & . \\ S_{m1}, \ldots, S_{mn} & \\ T_1, \ldots, T_m, O & \end{array} \begin{array}{l} T = D'\ \rightarrow\ O, \\ T_1 = O, \ldots, T_m = O, \\ S_{11} = F'_{11}, \ldots, S_{1a} = F'_{1a}, \\ \quad\quad\quad \vdots \\ S_{m1} = F'_{m1}, \ldots, S_{mn} = F'_{mn}, \\ E_1, \ldots, E_m \end{array} \right\rangle$$

4.2 Function defintions with an annotated type

These are the function definitions for which the programmer has annotated the expected type of the function. Once the function is type inferred, an attempt is made to unify the annotated function type with the inferred one. If the two types can be successfully unified, then the inferred type is line with the expectation of the programmer and the annotated type is inserted in the symbol table with the name of the function.

The type equations generated for the annotated functions differ slightly from the unannotated

ones. In the interest of succinctness, we will describe the equation generation only for the variable patterns case. Equation generation for the constructor pattern case can be easily extrapolated from this.

Let the annotated function be represented as following.

$$annotatedType = \forall\ A_1, \ldots, A_k\ .fType$$

Renaming the variables A_1,...,A_k with A'_1,...,A'_k yields the following type.

$$\begin{aligned} annotatedType' &= (\Lambda A_1, \ldots, A_k\ .\ annotatedType)\ A'_1, \ldots, A'_k \\ &= \forall\ A'_1, \ldots, A'_k\ .fType' \end{aligned}$$

Typing rule for function defns with variable patterns

$$\frac{\Gamma \vdash \begin{array}{l}\mathsf{fun\ f} \\ \quad \mid\ x_1, \ldots, x_m \ \rightarrow\ t\end{array} : T \ \langle E_{fun} \rangle}{\Gamma \vdash \begin{array}{l}\mathsf{fun\ f} :: \forall\ A'_1, \ldots, A'_n\ .fType' \\ \quad \mid\ x_1, \ldots, x_m \ \rightarrow\ t\end{array} : T \ \left\langle \begin{array}{lll} \forall\ A'_1, \ldots, A'_n & & T = fType', \\ \exists\ T & . & E_{fun} \end{array} \right\rangle}\ \text{fdefn}_1$$