Typing Rules for Sequential MPL

April 19, 2017

1 Type Inferencing

1.1 Introduction

This chapter deals with the type inference of the various MPL constructs, namely functions, processes, terms and process commands.

1.2 Type Formation Rules

Let T be a set of atomic types, Ω_D be a set of type formation symbols for data types and ω_D is a function called arity that describes the number of parameters that the type needs.

$$\omega_D \ : \ \Omega_D \ \rightarrow \ \mathbb{N}$$

Similarly, ω_C is the arity function for Ω_C which is a set of set of type formation symbols for codata types.

$$\omega_C \ : \ \Omega_C \ \rightarrow \ \mathbb{N}$$

Elements of T can be type variables or type constants like Int, Double and Char. Apart from the atomic types, MPL also has data types, codata types and products as types. Type formation rules describes the rules for the construction of valid types from the elements of T.

The type formation rules for MPL are given by the following rules.

Type Formation Rules

$$\frac{A \in T}{A \quad type}$$

$$\frac{A_1\ type \qquad \ldots \qquad A_n\ type}{A_1 \times \ldots \times A_n \quad type}\ prod$$

$$\frac{A_1\ type \qquad \ldots \qquad A_n\ type \qquad \omega(D) = n}{D\ (A_1 \ldots A_n) \quad type}\ data\ type$$

$$\frac{A_1\ type \qquad \ldots \qquad A_n\ type \qquad \omega(C) = n}{C\ (A_1 \ldots A_n) \quad type}\ codata\ type$$

1.3 Type Inferencing

Consider an Algebraic Type System consisting of functions and variables as terms. The term formation rules for the terms are as below.

$$\frac{}{\Gamma, x : A \vdash x : A}\ \text{var}$$

$$\frac{\Gamma \vdash m_1 : A_1, \qquad \ldots \qquad \Gamma \vdash m_k : A_k \qquad f : A_1, \ldots, A_k \to B \in \Omega}{\Gamma \vdash\ f(m_1, \ldots, m_n) : B}\ \text{fun}$$

In this type system consider a function, $f(x, y) = x$ such that $f : A, B \to A \in \Omega$. Let us try inferring the type of $f\ (1, y)$.

$$\frac{y : B \vdash y : B \qquad y : B \vdash 1 : Int \qquad f : A, B \to A \in \Omega}{y : B \vdash\ f(1, y) : Int}$$

Please note that $f : Int, B \to Int$ is a more specific type than the type of f initially assigned in Ω. The new more specific type of f is consistent with its original more generic type. However, this way of type inferencing while suitable for calculation by hand doesn't lend itself well to automation. For this reason, the typing rules are enhanced with equations that capture the constraint relationship between the different parts of the construct being type inferred.

$$\frac{}{\Gamma, x : A \vdash x : B\ \ \langle A = B \rangle}\ \text{var}$$

$$\frac{\Gamma \vdash m_1 : A'_1, \qquad \ldots \qquad \Gamma \vdash m_k : A'_k \qquad f : A_1, \ldots, A_k \to B \in \Omega}{\Gamma \vdash\ f(m_1, \ldots, m_n) : C\ \ \langle B = C, A_1 = A'_1, \ldots, A_k = A'_k \rangle}\ \text{fun}$$

Now consider type inferring the function $f(1, y)$ in this setup.

$$\frac{y : B \vdash y : B' \qquad y : B \vdash 1 : Int \qquad f : A', B' \to A \in \Omega}{y : B \vdash\ f(1, y) : Int \quad \langle A' = Int,\ B' = B,\ A = Int \rangle}$$

Once the type equations are solved by relacing the value of A, B and A' in function type of f, $Int, B \rightarrow Int$ is obtained. This is the type of function f.

The type inference has been broken down into two mechanical steps, namely *Generation of Type Equations* and *Solving Type Equations*. The mechanical and algorithmic nature of this presentation is useful when implementing type systems on a computer.

One drawback of the equational presentation of the type systems mentioned in the previous paragraph is that one ends up with a big linear list of equations. Solving this list of equations maybe slow as one needs to go through the entire list every time in order to look for substitutions.

In this thesis, we have modified this presentation to take advantage of the structure of the *type inference tree* when searching for substitutions. In this approach, the algorithm only searches the part of the type inference tree which could have generated these substitutions. Thus our algorithm localises the substitution search leading to faster soltuion of type equations. Our algorithm also has an added advantage of localising the type errors. This results in better location accuracy when reporting the type errors.

Consider a function $f(a, b)$ such that $f: \ A, B \rightarrow C \in \Omega$ and one needs to type infer $f(x + y, z)$ in our setup. In the below *type inference tree*, the context x:X,y:Y,z:Z has been replaced with symbol Γ for the purpose of succinct representation.

$$\dfrac{\dfrac{\overline{\Gamma \vdash x : D} \qquad \overline{\Gamma \vdash y : E}}{\Gamma \vdash x + y \ : C} \qquad \overline{\Gamma \vdash z : B} \qquad f : P \ , Q \rightarrow R \in \Omega}{\Gamma \vdash \ f(x + y, z) : A \quad \langle A' = Int, \ B' = B, \ A = Int \rangle}$$

1.4 Overview of Type Inference of MPL Constructs

Type inference involves two main steps.

- **Generating Type Equations** - Type Equations represent the constraint relationship between the different parts of the construct being type inferred.

  Using the typing rules, a series of equations are generated in the proof search direction. The equation structure hierarchical following 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. This approach allows hierarchical solution of type equations and better localisation of type errors.

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

In this chapter, the generation fo type equations for the Sequential MPL (terms and functions) is dicussed followed by the generation of type equation for Concurrent MPL. Finally, we look at an algorithm to solve the type equations to get the most general type.

2 Generating Type Equations for Sequential Terms

Here the type inference rules for the various sequential MPL constructs are dicussed. The type inference rules can be thought of as term formation rules enhanced with the typing information of the constituting terms.

2.1 Variable Term

In order to infer a variable which is a sequential term,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 variable
$\dfrac{}{x : P, \Gamma \vdash x : Q \quad \left\langle P = Q \right\rangle}$ id

2.2 Function Call

Let Ω be a set of function symbols and T_y be a type. σ is a function of the below type that returns as output the the input and output types of a function symbol.

sigma is defined both for inbuilt functions as well user defined functions. Example of σ for inbuilt functions is below.

$$\begin{aligned} \sigma(+_{Int}) &= ([Int, Int], Int) \\ \sigma('s') &= ([\,], Char) \end{aligned}$$

Consider a user defined function factorial which takes an integer as the input and outputs the factorial of the integer. The type of the output is also an integer. σ for factorial is defined below.

$$\sigma(fact) = ([Int], Int)$$

Consider a function f taking m inputs of types S_1, ... S_m. The output type of the function is S.

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

The output and input types are α-renamed before using these constructs in the type equations. This is done to avoid any name clashes between the variables of the function and the variables already used in the equation.

$$\begin{aligned} S'_i &= (\Lambda A_1, \ldots, A_k.\ S_i)\ A'_1, \ldots, A'_k \quad \{\ 1\ \leq\ i\ \leq m\} \\ S' &= (\Lambda A_1, \ldots, A_k.\ S)\ A'_1, \ldots, A'_k \end{aligned}$$

- The type of a function call is the output type of the function type of the function being called.
- The type of the various input arguments of the function call can be deduced from the input types of the function type of the function being called.

Typing rule for function call

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

2.3 Constant Term

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

Typing rule for constants

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

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

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

2.4 Case and Constructor Term

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

case term branches on the different constructors of a data type. Every branch consists of a constructor and a term as can be seen from the case syntax on page 7. The term is executed when the corresponding constructor is selected.

2.4.1 Data Type Declaration

Let us define a data type D(A_1, ..., A_k) and C_1 ... C_m be its data constructors. The data type D (A_1, ..., A_k) is polymorphic in type variables A_1, ..., A_k which means that A_1, ..., A_k are the

union of type variables used in the different constructors of the data type. For constructor C_1 which takes **"a"** number of input terms, F_{11}, ..., F_{1a} repesent the input types of those terms.

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

The different constructs of the data type definition above are α-renamed with fresh variable A'_1, ..., A'_k to avoid any naming conflicts when these constructs are used in type equations. For the same reason, we will use the α-renaming technique while type inferring other MPL constructs as well. The renamed constructs have been represented by adding a superscript $\prime$ to the name of their names.

Λ is a function that takes as arguments a data construct, the old variables present in that data type construct and the fresh variables to replace the old variables. The output of this function is a data type construct where the old variables in its body have been replaced by the new variables.

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

2.4.2 Type Equations for Case and Constructor Term

The function type of a constructor can be looked up from the symbol table.

- The output type of the constructor term is the output type of the function for that constructor looked up from the symbol table.
- The types of the various arguments of the constructor are the corresponding input types for that constructor looked up from its function type from the symbol table.
- E_1, ..., E_j are the type equations generated for the terms t_1, ..., t_j 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 constructor and case

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

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

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

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 over which one is folding. Every branch has some argument on the left side of the arrow and a term on the right side as can be seen in the fold syntax on page 8.

For all the constructors of a data type, their fold functions are inserted in the symbol table. We will use the data definition for D(A_1, ..., A_k) and the renamed constructs from the section 2.4.1 on page 5

The type of a fold term is the type of the terms on the right side of the branches. In a well typed fold term, all the terms on the right side of the branches will have same type. 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

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

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

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.

2.6.1 Codata Type Declaration

Consider the codata declaration for a codata type $\mathrm{C}(A_1,\ldots,A_k)$. D_1, ..., D_m are the different destructors of the codata type. $\mathrm{C}(A_1,\ldots,A_k)$ is polymorphic in type variables $A_1,\ldots,A_k$ which means that $A_1,\ldots,A_k$ are the type variables used in the different destructors of the codata type. For destructor D_1 which takes **"a"** number of input terms , F_{11}, ..., F_{1a} repesent the types of the input terms.

$$\begin{array}{lll} \forall A_1,\ \ldots,\ A_k.\ codata\ A\ \to\ C(A_1,\ldots,A_k) = & D_1 : F_{11},\ \ldots,\ F_{1a},\ A \to P_1 \\ & \vdots \qquad\quad \vdots \qquad\quad \vdots \\ & D_m : F_{m1},\ \ldots,\ F_{mn},\ A \to P_m \end{array}$$

Renaming the different constructs in the codata $\mathrm{C}(A_1,\ldots,A_k)$ with fresh variables A'_1, ..., A'_k gives the following constructs.

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

2.6.2 Type Equations for Record and Destructor Term

The type of a record of a codata type is that codata type.

The function type of a destructor can be looked up from the symbol table. Following details about the type of destructor can be obtained from its function type.

- if a destructor takes n arguments, then the corresponding function type of the destructor will have (n+1) input types. The first n types out of the (n+1) types are the types of the corresponding arguments. The $(n+1)^{th}$ type is the codata type of the destructor.
- The output type of the destructor is the output type of the function for that destructor looked up from the symbol table.

Typing rules for record and destructor

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

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

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

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

2.7 Product Term

prod term is used to represent 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 If Term

if term has 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 the first and second statement. A correctly typed if term will have these two statements of the same type.

Typing rule for if

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

2.9 Let Term

let statment allows the programmer to define local variables and 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 type inferred and put in the symbol table. All that remains in the where part are local variable definitions.

The type the let statement is the type of the term $\mathbf{t}$. 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

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

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 and processes, 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.

Typing rule for constructor pattern

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

3.1.6 product pattern

Typing rule for product pattern

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

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.

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 function defns with variable patterns

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

$$\left\langle \exists\, S_1, \ldots, S_m \;.\; \begin{array}{l} T = (S_1, \ldots, S_m) \;\rightarrow\; O, \\ E \end{array} \right\rangle$$

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, ..., A_k are the union of type variables used in the different constructors of the data type.

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

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

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

Typing rule for function defns with constructor patterns

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

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

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

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

4.3 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

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