Prolog Type Checking Language (PTCL)
DSL Report Card

Ghadeer Al Kubaish, Qiaoran Li
School of EECS
Oregon State University

June 13, 2018

1 Introduction

Prolog, a logical programming language, has been widely used in artificial intelligence research. Its program is a set of facts and rules that represent the relationship between values. Prolog is also an untyped programming language, most of current existing type checkers for Prolog are basically implemented as semi-deterministic predicates[1] in Prolog, there is no specific DSL existing to perform type checking.

The Prolog Type Checking language (PTCL) provides type checking for Prolog which guarantees rules with type declaration in Prolog program are well-typed, where the type declaration describes the name of the predicate and the types of its arities and the user can also define their own types. A Prolog program is well-typed when the predicates are satisfied with type declaration. By using PTCL, users can prevent type error before running the program and feel easier to debug the program with error and warning messages.

2 Users

The PTCL is designed for Prolog programmers, especially for beginners.

3 Outcomes

The PTCL will serve the following outcomes:

(1) Allow defining new types in the Prolog using data constructors and type synonyms.
(2) Allow predicates type declarations.
(3) Find type errors in the Prolog program.
(4) Provide a detailed report of errors and warnings about ill-typed predicates and undetermined predicates.

4 Concrete Grammar

The main DSL elements expressed in the Prolog Type Checking language are user-defined type and type signature for rules. The concrete grammar of PTCL is shown as below:

```
                t ∈ Type ::= atom  | int  | string | tv | udt
     td ∈ TypeDefinition ::= data udt tv = constrs. | type nt = t.
  constrs ∈ Constructors ::= constr1 | constr2 | ... | constrn
    constrn ∈ Constructor ::= con(t*) | con
tdels ∈ TypeDeclarations ::= decl tdel*.
  tdel ∈ TypeDeclaration ::= pt | ft
      pt ∈ PredicateType ::= pn(t*).
        ft ∈ FunctorType ::= fn(t*) -> t.
                      con ∈ constructorName
                       tv ∈ TypeVariable
                      udt ∈ UserDefinedType
                      pn  ∈ PredicateName
                      fn  ∈ FunctorName
```

TypeDefinition defines a new type in Prolog program. It could either use **data** to introduce a new data type or **type** to create a new type synonym. To distinguish user defined type name with the type variable, the name of type variable should start with underscore '_'.
TypeDeclarations declares the type signature for predicates and functors. The declaration should start with the keywords **decl**. As for type signature for functors, the user should note the return type of functors in the right hand side of '→'.
Note that the PTCL adopts the convention of traditional Prolog, every user defined type and type declaration should end with '.'.

5 Use Cases / Scenarios

A well-typed program is the one that the entire prolog program satisfied the type declaration. If the prolog program failed to have same type with the type declaration, the PTCL will prompt the according error and warning message. This section illustrates the use cases of PTCL using examples.

5.1 Type Errors

There are several type error which can be prevented by the DSL shown as follows:

Unknown Types This is a scenario where the type used in the declaration is unknown. For example,

```
type myList = list.
decl listLength(myLi, int).
```

In this case, the type *myLi* here is not either a build-in type or a user defined type. The error message from PTCL should be:

```
- Unknown User-Defined-Type "myLi"
- Declared at: listLength("myLi",int)
```

Overloading Declaration This is a scenario where there exists overloading of a predicate name. For example,

```
decl female(atom).
decl female(string).
```

will trigger the *Overloading Declaration* error, since there are two type declarations for the same predicate name *female*, which is prohibited by our type system. The error message from PTCL should be:

```
- Overloading declaration of female
- Declared at
- female(atom)
- female(string)
```

Overloading Definition This is a scenario where there exists an overloading of a user-defined type name. For example,

```
data tree _a = node(_a, tree _a, tree _a ) | leaf.
data tree = node(string, tree,tree ) | leaf.
```

will trigger the *Overloading Definition* error, since there are two types with the same name *tree* which is prohibited by our type system. The error message from PTCL should be:

```
- Overloading definition of tree
- Defined at:
- data tree ( "_a" ) = node(_a,"tree" "_a","tree" "_a") | leaf
- data tree = node(string,"tree","tree") | leaf
```

Overloading Constructor Name This is a scenario where there exists am overloading of user-defined type constructor name. For example,

```
data tree _a = node(_a, tree _a, tree _a ) | leaf.
data tree = node(string, tree,tree ) | leaf.
```

will trigger the *Overloading Constructor Name* error, since there are multiple constructors with the same name, *node* and *leaf* which is prohibited by our type system. The error message from PTCL should be:

```
- Overloading constructor name node
 - In data tree ( "_a" ) = node(_a,"tree" "_a","tree" "_a") | leaf
 - In data tree = node(string,"tree","tree") | leaf

- Overloading constructor name leaf
 - In data tree ( "_a" ) = node(_a,"tree" "_a","tree" "_a") | leaf
 - In data tree = node(string,"tree","tree") | leaf
```

Misusing *is* The scenario happens when the user misuses the reserved words in Prolog program. For example, we define an arithmetic notation to manipulate integers. The correct notation should be

```
8 is 5+3.  %correct demo
```

Here, the right-hand side of *is* should be the expression, and the left-hand side of *is* should be the number that the right-hand expression evaluates to. There are common errors when dealing with *is* such as reversing the expressions of the left and right hand sides of *is*. For example,

```
double(X, Y) :- X * 2 is Y.
```

In this case, the PTCL should produce error message:

```
- Misuse of "is": expecting Number in the left hand side of "is"
- In "X" * 2 is "Y"
- In double("X","Y") :- "X" * 2 is "Y".
```

Incorrect Argument Type This is a scenario where the type of the predicates argument does not satisfy with that of type declaration.

example 1:

```
decl age(atom, int).
decl eq(_a,_a).

ege6(X) :- age(X,Y), eq(X,6).
```

will trigger the *Incorrect Argument Type* error, since the expected type for predicate *eq* was *(_a,_a)* while it was passed two different types. We substitute *_a* with *atom* and then we try to substitute it with *int*, so it fails. In this case, the error message from PTCL should be:

```
- Couldn't match expected type eq(atom,atom) with eq(atom,int)
- In eq("X",6)
- In ege6("X") :- age("X","Y") , eq("X",6).
```

example 2:

```
decl sumTree(tree, int).

sumTree(node(I, L, R), T ):- sumTree(L, N1), sumTree(R, L), T is N1 + I.
```

will trigger the *Incorrect Argument Type* error, since the expected type for predicate *sumTree* was *(tree _a,int)* while it was passed two trees in the second body element *sumTree(R, L)*. In this case, the error message from PTCL should be:

```
- Couldn't match expected type sumTree("tree" "_a",int) with sumTree("tree" "_a","tree" "_a")
- In sumTree("R","L")
- In sumTree(node("I","L","R"),"T") :- sumTree("L","N1") , sumTree("R","L") , "T" is "N1" + "I".
```

example 3:

```
decl mytree(tree _a ).

mytree(node(4, leaf, node(3,leaf(4), node(4,leaf, leaf)))).
```

will trigger the *Incorrect Argument Type* error, since the expected type for predicate *mytree* was *tree _a* while it was a value of incorrect form. The constructor *leaf* was giving an arguemnt of type *int* and it is expecting no arguemnts which makes this error. In this case, the error message from PTCL should be:

```
- Couldn't match expected type myTree("tree" "_a")
with myTree(node(int,tree "_a",node(int,leaf(int),node(int,tree "_a",tree "_a"))))
- In myTree(node(4,"leaf",node(3,leaf(4),node(4,"leaf","leaf"))))
- In myTree(node(4,"leaf",node(3,leaf(4),node(4,"leaf","leaf")))).
```

Conflicting Types For Comparison This is a scenario where the type in the left and right hand side of $=$ or $/=$ are different. Other comparison operators should have *int* type in both sides.

example 1:

```
decl isEqual(int, atom).

isEqual(X,Y) :- X = Y.
```

will trigger the *Conflicting Types For Comparison* error, since we were expecting the same type in both sides, but were giving two types *int* and *atom*. In this case, the error message from PTCL should be:

```
- The types "int" and "atom" do not match
- In: "X" = "Y"
- In isEqual("X","Y") :- "X" = "Y".
```

example 2:

```
decl formTree(a,b,tree).

formTree(X,Y, nodeAB(X,Y,leafAB(Y))) :- X = 5 , Y = "S".
```

will trigger the *Conflicting Types For Comparison* error, since the expected type for predicate *formTree* was *(_a, _b,tree _a _b). _b* was substituted with *tree _a _b* and *_a* was substituted with *int*. Then we try to unify *_b* which has type *tree _a _b* with *string*, so we get a conflict. In this case, the error message from PTCL should be:

```
- The types ""treeAB" "_a" "_b"" and "string" do not match
- In: "Y" = "S"
- In formTree("X","Y",nodeAB("X","Y",leafAB("Y"))) :- "X" = 5 , "Y" = "S".
```

Variable Type Error This is a scenario where the type of a variable conflicts with the expected type.
example 1:

```
decl sumTree(tree, int).

sumTree(node(I, L, R), T ):- sumTree(L, N1), sumTree(R, N2), T is N1 + R + I.
```

will trigger the *Variable Type Error* error, since we were giving *R* type *tree _a* and we are using it as *int* in *T is N1 + R + I* which produces conflicting types. The error message from PTCL should be:

```
- Couldn't match expected type int with R:"tree" "_a"
- In "T" is "N1" + "R" + "I"
- In sumTree(node("I","L","R"),"T") :- sumTree("L","N1") ,
    sumTree("R","N2") , "T" is "N1" + "R" + "I".
```

example 2:

```
decl formTree(a,b,tree).

formTree(X,Y, nodeAB(X,leafAB(Y),leafAB(Y))) :- X = 5 , Y = "S", X is Y.
```

will trigger the *Variable Type Error* error, since the expected type for predicate *formTree* was *(_a, _b,tree _a _b). _*a was substituted with *int* and *_b* was substituted with *string*. Then we used them in *Y is X* which expects two ints, so we try to substitute with *int* and get a conflict. In this case, the error message from PTCL should be:

```
- Couldn't match expected type int with Y:string
- In "Y" is "X"
- In formTree("X","Y",nodeAB("X",leafAB("Y"),leafAB("Y"))) :- "X" = 5 ,
    "Y" = "S" , "Y" is "X".
```

Incorrect Arities This is a scenario where the arities of a predicate do not satisfy with type declaration. For example,

```
decl age(atom,int).

age(jacki, 19, 19).
```

will trigger the *Incorrect Arities* error, since the user used 2 argument in its predicate while we expected one. In this case, the DSL should produce the error message informing the user about the incorrect arities. The error message from PTCL should be:

```
- The predicate for age(atom,int) expect 2 arguments, but age(atom,int,int) has 3 arguments.
- In age("jacki",19,19)
- In age("jacki",19,19).
```

5.2 Warnings

List of Non-declared predicates The PTCL will provide some warning message about predicates which do not have a corresponding type declaration. For instance,

```
decl female(atom).

female(mona).
male(justin).
child(ruby).
```

Here, there is no type declaration of the predicates *male()* and *child()*, in this case, the PTCL will provide a list of such predicates in warning messages as below:

```
- Non-Declared: male("justin")
- Non-Declared: child("ruby")
```

Conflicting argument type This warning happens when a predicate is not declared (which is allowed) but then used with two different arguments' types. Since Prolog allowed such situation, but User may overload predicate name by mistake, PTCL should provide warning message to inform user. For instance,

```
male(5).
male(jack).
```

Here, there is no type declaration for male, though, the user has different argument's type for *male()*, that is, *int* and *atom*. In this case, the PTCL will provide a list of such predicates in warning messages as below:

```
- Non-Declared: male(5)
- Non-Declared: male("jack")
- conflicting argument type: male(5) and this male("jack")
- conflicting argument type: male("jack") and this male(5)
```

6 Basic Objects

The Basic Objects in PTCL are composed of three main parts, that is, user defined types, type declarations and Prolog program.

6.1 Types

Type: There are 6 types in the program: Atom, Int, String, List, TypeVar, and User defined type.

```
data Type = TAtom
          | TInt
          | TString
          | TList
          | TVar TypeVar
          | TDef TypeName [TypeVar]
```

User Defined Type: In PTCL, the user can define their own types in the program. User defined types can either be a type synonym or a data type constructor. The first part of our Prolog file should contain the new type definitions that the user created.

```
type Line = Int
type TypeDef = [(DefinedType, Line)]
data DefinedType = TypeT TypeName Type
                 | DataT TypeName [TypeVar] [Cons]
type Cons = (ConstructorName,[Type])
type TypeName = String         -- lower case
type TypeVar = String          -- lower case start with _
type ConstructorName = String  -- lower case
```

Type Declarations: The section part of the Prolog file should contain a list of type declarations. Each declaration is represented by the name of the predicate associated with type of its arguments. Type dictionary is a list of type declarations. We use these declarations to make sure the predicates definitions that the user writes later match the type declarations. We only declare functors and predicates. Functors have a return type. We can also think of user-defined types is functors which have the constructed type as their return type.

```
type TypeDic = [(Dec,Line)]
type PredicateT = (PredName, [Type])
type FunctorT =   (FuncName,[Type],Type)
data Dec = PredD PredicateT | FuncD FunctorT
```

6.2 Prolog Program

The last part of the Prolog file should contain the rules of the program (clauses). This part is what we want to use the DSL to type check it. The following are the types we need to construct the AST of the program.

Names: Name of variable, name of atom and name of predicate.

```
type VarName = String  -- upper case
type AtomName = String -- lower case
type PredName = String -- lower case
type FuncName = String -- lower case
```

Predicate: The predicate consists of two parts: a predict name and a list of arguments. Each argument is represented by a constructor of its type binding with its value. The argument is a constant value, a variable, an arithmetic expression or a functor.

```
data Argument = Atom AtomName
              | LitI Int
              | LitS String
              | List [Argument]
              | Var VarName
              | Func PredFunA
              | OperA OptA Argument Argument
```

Rule: Each line in the Prolog program contains a rule, which consists of a head and a body. The head contains a predicate and the body is either empty such as in facts or contains other body elements. Body elements include these elements: another predicate definition, an "is" expression, a Boolean expression, or and of two predicates.

```
type PredFunA = (String, [Argument])
data Rule = Head PredFunA Body
type Body = [BodyElem]
data BodyElem  = Pred PredFunA
     | Is Argument Argument
     | OperC OptC Argument Argument
     | And BodyElem BodyElem
```

Operations: A set of arithmetic and comparison operations.

```
data OptA =  Sub | Add | Div | Mult| Mod
data OptC =  Eq | Neq  | Lt | Leq | Gt | Gtq
```

7 Operators and Combinators

To perform type checking in PTCL, there are 2 phases that need to act, that is, transforming concrete syntax (in Section 4) into abstract syntax (in Section 6), and throwing the abstract syntax tree into the type checker to reason about the types of Prolog program.

In phase one, we parse the Prolog program using a Haskell package named Megaparsec[2] which let us compose higher order functions to generate parsers.

The parser definition is:

```
type Parser = Parsec Void String
```

The main combinators used to build the parser for PTCL as below:

```
many   :: Parser a -> Parser [a]
sepby  :: Parser a -> Parser b -> parser [a]
try    :: Parser a -> Parser a
option :: a -> parser a -> parser a
```

where *many* is used to parse repeated applications of a parser; *sepby* is used to parse repeated applications of the first parser separated by the second parser; A parser *try p* backtracks the parser state when *p* fails; *option a b* gives one more choice for parser, that is, if parser *b* failed, the result will return *a*;

In phase two, we implement a type checker for type checking the Prolog program. There are 3 main functions to do the type-checking task.

```
errors   :: TypeDef -> TypeDic -> Prog -> Maybe [Err]
warnings :: Prog -> TypeDic ->  TypeDef -> Maybe [War]
checker  :: (TypeDef,TypeDic, Prog) -> Report
```

The *errors* function takes the type definitions, type declaration, and prolog program as arguments and provides the error report or nothing. The *warnings* function takes the prolog program, type declaration and type definitions and provides the warning report or nothing. Then, the *checker* will combine the errors and warnings to provide the final report.

8 Implementation Strategy

The typing rules explains what we are doing in a high level. The following typing rules are referred to from Mycroft and A O'Keefe's paper. [3]

First we have these assumptions

Q	clause C $\leftarrow B_1,..,B_m$
P	a finite subset of Var $\cup$ Pred $\cup$ Functor
$\hat{P}$	an association of an extended type of each symbol in Q
σ_i and τ	represent non extended types
	X^τ is variable X of type τ
$a(\sigma_1,..\sigma_k)$	predicate a of arity k
$f((\sigma_1,..\sigma_k) \rightarrow \tau)$	functor f of arity k and return type τ
$\hat{Q}$	typed clause by writing the type of each of each term

If we have a rule (a predicate with body), which contains a head and a body, then the predicate must match the declaration that we have in the environment. Each of its argument must type check based on the environment and each of its body elements must also type check based on the environment.

$$\hat{P} \vdash (\text{A} \leftarrow B_1,...B_m \text{) if}$$
$$\text{A} = a({t_1}^{\tau_1},...{t_k}^{\tau_k}) \text{ and } a^\rho \in \hat{P}$$
$$\text{With } (\tau_1,...\tau_k) \cong \rho$$
$$\text{and } \hat{P} \vdash {t_i}^{\tau_i} \ (1 \leq i \leq k)$$
$$\text{and } \hat{P} \vdash B_i \ (1 \leq i \leq m)$$

If we have a functor, then the functors arguments must match the type declaration of that functor in the environment and its return type matches the what we are expecting the environment.

$$\hat{P} \vdash u^\sigma \text{ if}$$
$$\text{u} = f({t_1}^{\tau_1},...{t_k}^{\tau_k}) \text{ and } f^\rho \in \hat{P}$$
$$\text{With } ((\tau_1,...\tau_k) \rightarrow \sigma) \leq \rho$$
$$\text{and } \hat{P} \vdash {t_i}^{\tau_i} \ (1 \leq i \leq k)$$

For the (Is l r) case, the left and right sides must be of type int and the expression must match the environment (no conflicting types).

$$\hat{P} \vdash \text{(Is l r) if}$$
$$\text{l} = {l'}^{Int}$$
$$\text{r} = {r'}^{Int}$$
$$\hat{P} \vdash {l'}^{int}$$
$$\hat{P} \vdash {r'}^{int}$$

For the (Com (= , $\neq$) l r) case, the left and right sides must have the same type and the expression must match the environemnt (no conflicting types).

$$\begin{gathered}\hat{P} \vdash \text{(Com op l r) if} \\ \text{op is = or } \neq \\ \text{l} = l'^{\tau} \\ \text{r} = r'^{\tau} \\ \hat{P} \vdash l'^{\tau} \\ \hat{P} \vdash r'^{\tau}\end{gathered}$$

For the (Com $(<, >, <=, >=)$ l r) case, the left and right sides must be of type int and the expression must match the environemnt (no conflicting types).

$$\begin{gathered}\hat{P} \vdash \text{(Com op l r) if} \\ \text{l} = l'^{Int} \\ \text{r} = r'^{Int} \\ \hat{P} \vdash l'^{int} \\ \hat{P} \vdash r'^{int}\end{gathered}$$

Lastly, if we have a variable then it must exist in the environment after applying the substitution.

$$\hat{P} \vdash X^{\sigma} \text{ if } X^{\sigma} \in \hat{P}$$

For the substitution strategy, if we found a variable and we can not refer to its type from the environment (no declaration), we give it a type variable. If we then found a type for that same variable, we substitute the type variable with that type. If we had conflicting substitutions, then there is an error. We use our "is" and comparison expressions to infer types when a declaration does not exist.

In an implementation level, we constructed a detailed report or warnings and errors. We have a function that seeks every error type. Some error types such as incorrect arities and incorrect argument types are combined in one method. When also maintain a map of variables to types and a map of type variables to substituted types to have more expressive error report.

```
data Warining = NonDecl PredFunA Bool
              | Conflict PredFunA PredFunA

data Error = ArgType Dec PredFunA VarMap
           | IncArrit Dec PredFunA VarMap

           | VariableType BodyElem Type VarMap
           | EqType BodyElem Type Type VarMap
           | MissIs  BodyElem

           | UnknowType TypeName Dec VarMap
           | MultDec Dec Dec VarMap
           | MultCon ConstructorName DefinedType DefinedType
           | MultDef DefinedType DefinedType

type VarMap = (VarTypes,Subsitutions)

data Err = E Line Error
data War = W Line Warining
```

9 Related DSLs

There has been many research on integrating type system on traditionally untyped Prolog. To our knowledge, [3] first introduced polymorphic type scheme for Prolog, which makes static type checking in Prolog possible. Based on theory of it, several strongly typed Prolog variants such as Mercury, Godel, Visual Prolog and Ciao have proposed.

Except for those standalone typed Prolog system, Towards Typed Prolog[4] proposes to build a type system as an add-on library to combine typed and untyped Prolog program. Towards Typed Prolog gradually proposes a type system in the style of standard Hindley-Milner, the basic type system is composed of types, type definition and type signature. Their system supports many build-in Prolog features such as arithmetic expression and meta-predicates. It also introduces an approach to interface typed and untyped prolog. But this paper just make proposals for having type checking in Prolog, to our best knowledge, they didn't implement the theory of paper.

PTCL actually implement an operating type checker (and parser) for Prolog. Besides that, we design a domain specific language to allow user define their own type and declare the type annotation. Nevertheless, PTCL provides a detailed report of errors and warnings about ill-typed predicates and undetermined predicates.

10 Future Work

The PTCL is expressive to accommodate the commonly type errors for Prolog. We did encounter some limitations of our language and will improve them in our future work:

Polymorphic: PTCL works good if the type variables occur in the declarations such as

```
decl eq(a,b).
```

However, if we no nothing about a variable then we are limited because we did not implement the algorithm that assigns type variables correctly to variables. For example,

if we know nothing about this predicate

```
x(T,Y):- ....
```

We are not able to correctly assign *T* to have type variable *_a* and *Y* to have type variable *_b*. We currenly give both *Y* and *T* the type varibale *_a*.

Type Inference for Warning: The warnings report could be improved using type inference. Right now, we do not infer types in the process of providing warnings.

Detailed Syntax Error Message: Although PTCL provides a detailed error report for the type errors, for the syntax error, we still use the naive error message from Parser. In the future work, we plan to add more readable and custom error messages for users.

Efficiency: We also think that our type checker could be improved using higher order functions to minimize the amount of code we wrote. Also, we might be able to improve the amount of search we perform by apply all of our error checking functions in one rule before moving to the other.

References

[1] SWI-Prolog Manual. http://www.swi-prolog.org/pldoc/man?section=typetest

[2] Megaparsec. http://hackage.haskell.org/package/megaparsec

[3] Alan Mycroft and Richard A. O'Keefe. *A polymorphic type system for prolog*. Artificial Intelligence, 23(3):295-307,1984.

[4] Tom Schrijvers, Vitor Santos Costa, Jan Wielemaker, and Bart Demoen. *Towards Typed Prolog*. In Proc. of the International Conference on Logic Programming (ICLP '08). Lecture Notes in Computer Science, vol. 5366. Springer, 693697.

Appendix A More Examples with Error Report

```
data tree _a = node(_a,tree _a ,tree _a ) | leaf.
type myList = list.
data tree = node(string,tree ,tree ) | leaf.
data treeAB _a _b = nodeAB(_a,treeAB _a _b,treeAB _a _b) | leafAB(_b).

decl female(atom).
decl female(string).
decl age(atom,int).
decl married_(atom, atom).
decl married(atom, atom).
decl myTree(tree _a ).
decl isTree(tree _a).
decl sumTree(tree _a , int).
decl listLength(myLi, int).
decl eq(_a, _a).
decl eq6(int).
decl isEqual(int, atom).
decl eq2(_a, _b).
decl comTwo(_a, _b).
decl formTree(_a, _b, treeAB _a _b).
decl doubleAge(atom, int).
```

1. The program.

```
male(mona).
male(5).
male(marge).
male([8]).

age(mona, 6).
age(jacki, 19, 19).
age(20,marge).

married_(abe,mona).
married_(clancy,jackie).
married_(homer,marge).

married(X,Y) :- married_(X,Y).
married(X,Y) :- married_(Y,X).

eq(X, Y) :- X = Y.
ege6(X) :- eq(6,X).
ege6(X) :- age(X,Y), eq(X,6).
ege6(X) :- age(X,Y), eq2(X,6).

isEqual(X,Y) :- X = Y.
comTwo(X,Y) :- X = 5 , age(X,Y).
double(X, Y) :- X * 2 is Y.
doubleAge(A,T):-  age(Y,A) , T is A * 2.
```

```
myTree(leaf).
myTree(node(leaf , leaf, leaf)).
myTree(node(4, leaf, node(3,leaf(4), node(4,leaf, leaf)))).

isTree(leaf).
isTree(node(_, L, R)):- isTree(L),isTree(R).

sumTree(0, leaf).
sumTree(node(I, L, R), T ):- sumTree(L, N1), sumTree(R, L ), T is N1 + I.
sumTree(node(I, L, R), T ):- sumTree(L, N1), sumTree(R, N2), T is N1 + R + I.

formTree(X,Y, nodeAB(X,leafAB(Y),leafAB(Y))) :- X = 5 , Y = "S", X is Y.
formTree(X,Y, nodeAB(X,Y,leafAB(Y))) :- X = 5 , Y = "S".

listLength([], 0).
listLength([_|T], Total):-  listLength(T, N) , Total is 1 + N.
```

2. According Error Report.

```
** Errors **
Line 1: - Overloading constructor name node
 - In data tree ( "_a" ) = node(_a,"tree" "_a","tree" "_a") | leaf
 - In data tree = node(string,"tree","tree") | leaf

Line 1: - Overloading constructor name leaf
 - In data tree ( "_a" ) = node(_a,"tree" "_a","tree" "_a") | leaf
 - In data tree = node(string,"tree","tree") | leaf

Line 15: - Unknown User-Defined-Type "myLi"
- Declared at: listLength("myLi",int)

Line 1: - Overloading definition of tree
- Defined at:
- data tree ( "_a" ) = node(_a,"tree" "_a","tree" "_a") | leaf
- data tree = node(string,"tree","tree") | leaf

Line 7: - Overloading declaration of female
- Declared at
- female(atom)
- female(string)

Line 31: - The predicate for age(atom,int) expect 2 arguments,
but age(atom,int,int) has 3 arguments.
- In age("jacki",19,19)
- In age("jacki",19,19).

Line 32: - Couldn't match expected type age(atom,int) with age(int,atom)
- In age(20,"marge")
- In age(20,"marge").
```

```
Line 44: - Couldn't match expected type eq(atom,atom) with eq(atom,int)
- In eq("X",6)
- In ege6("X") :- age("X","Y") , eq("X",6).

Line 47: - The types "int" and "atom" do not match
- In: "X" = "Y"
- In isEqual("X","Y") :- "X" = "Y".

Line 49: - Couldn't match expected type age(atom,int) with age(int,_b)
- In age("X","Y")
- In comTwo("X","Y") :- "X" = 5 , age("X","Y").

Line 51: - Misuse of "is": expecting Number in the left hand side of "is"
- In "X" * 2 is "Y"
- In double("X","Y") :- "X" * 2 is "Y".

Line 52: - Couldn't match expected type age(atom,int) with age(atom,atom)
- In age("Y","A")
- In doubleAge("A","T") :- age("Y","A") , "T" is "A" * 2.

Line 52: - Couldn't match expected type int with Y:atom
- In "T" is "A" * 2
- In doubleAge("A","T") :- age("Y","A") , "T" is "A" * 2.

Line 56: - Couldn't match expected type myTree("tree" "_a") with
myTree(node(int,"tree",node(int,leaf(int),node(int,"tree","tree"))))
- In myTree(node(4,"leaf",node(3,leaf(4),node(4,"leaf","leaf"))))
- In myTree(node(4,"leaf",node(3,leaf(4),node(4,"leaf","leaf")))).

Line 61: - Couldn't match expected type sumTree("tree" "_a",int) with sumTree(int,"tree")
- In sumTree(0,"leaf")
- In sumTree(0,"leaf").

Line 62: - Couldn't match expected type sumTree("tree" "_a",int) with
sumTree("tree" "_a","tree" "_a")
- In sumTree("R","L")
- In sumTree(node("I","L","R"),"T") :- sumTree("L","N1") , sumTree("R","L") , "T" is "N1" + "I".

Line 63: - Couldn't match expected type int with R:"tree" "_a"
- In "T" is "N1" + "R" + "I"
- In sumTree(node("I","L","R"),"T") :- sumTree("L","N1") , sumTree("R","N2")
, "T" is "N1" + "R" + "I".

Line 65: - Couldn't match expected type int with Y:string
- In "X" is "Y"
- In formTree("X","Y",nodeAB("X",leafAB("Y"),leafAB("Y"))) :- "X" = 5 , "Y" = "S" , "X" is "Y".

Line 66: - The types ""treeAB" "_a" "_b"" and "string" do not match
- In: "Y" = "S"
- In formTree("X","Y",nodeAB("X","Y",leafAB("Y"))) :- "X" = 5 , "Y" = "S".

Line 68: - Couldn't match expected type listLength("myLi",int) with listLength(list,int)
- In listLength([],0)
- In listLength([],0).
```

```
Line 69: - Couldn't match expected type listLength("myLi",int) with listLength(list,var)
- In listLength(["_" , "T"],"Total")
- In listLength(["_" , "T"],"Total") :- listLength("T","N") , "Total" is 1 + "N".

Line 69: - Couldn't match expected type listLength("myLi",int) with listLength(var,var)
- In listLength("T","N")
- In listLength(["_" , "T"],"Total") :- listLength("T","N") , "Total" is 1 + "N".
```

3. According Warning Report.

```
** Warnings **
Line 25: - Non-Declared: male("mona")
Line 26: - Non-Declared: male(5)
Line 27: - Non-Declared: male("marge")
Line 28: - Non-Declared: male([8])
Line 43: - Non-Declared: ege6("X")
Line 44: - Non-Declared: ege6("X")
Line 45: - Non-Declared: ege6("X")
Line 51: - Non-Declared: double("X","Y")
Line 28: - conflicting argument type: male("mona") and this male([8])
Line 26: - conflicting argument type: male("mona") and this male(5)
Line 27: - conflicting argument type: male(5) and this male("marge")
Line 25: - conflicting argument type: male(5) and this male("mona")
```