A Formal Type Inferencer for System-F in Lean

Yifan Song
Delft University of Technology
Delft, Netherlands
Y.Song-31@student.tudelft.nl

***Abstract*—**

I. Introduction

A type system is a common tool for verifying the correctness of a program, while polymorphism is a common way for both making the type system stronger and enabling us to write cleaner codes. as the type system becomes more complicated, we would want to have a type inferencer so that we need not write all type annotations explicitly. There are some common polymorphism ways like let polymorphism and system F. In this project, we consider system F which is also the foundation of the type system of Haskell and OCaml. A type inferencer for a strong system F might not be decidable, but with type annotation, we can give an inferencer for a weak system F.

In this project, a formalization of type checker for a simple type lambda calculus is given. Based on that system F is also implemented with a formalized type inferencer.

II. STLC with Recursive Function

A. Definition of Abstract Syntax, Type and Context

The syntax, type, and context of simple typed lambda calculus(STLC) with recursive function are defined the same as in the project description. The only difference is the added $@[derive\ decidable\_eq]$ notation. This is to tell lean that the comparison of whether two types are equal is decidable. So it is essential for the type checker to be a decidable algorithm.

B. Type Relations for STLC

Before implementing and formalizing the type inferencer for STLC, we need to implement the type relations for STLC so that we can type annotate an expression and judge the type inferencer afterward.

The type relations for literal terms are straightforward, taking expression ETrue as an example:

$$\frac{}{\vdash ETrue : TBool} \tag{1}$$

In the corresponding lean code, this is represented as '$True\_typed\ \Gamma : ctx : typed\ \Gamma\ exp.ETrue\ ty.TBool$' This is basically the same for $IsZero$, $Pred$ and $Succ$:

$$\frac{}{\vdash EIsZero : TFun(TNat, TBool)} \tag{2}$$

$$\frac{}{\vdash EPred : TFun(TNat, TNat)} \tag{3}$$

$$\frac{x : T \in \Gamma}{\Gamma \vdash EVar(x) : T} \tag{4}$$

For the Var expression, we need to check its type in the given environment, then get the type of this expression.

$$\frac{\Gamma \vdash e1 : TFun(T1, T12) \quad \Gamma \vdash e2 : T1}{\Gamma \vdash EApp(e1, e2) : T_{12}} \tag{5}$$

Here we need to introduce the type of the two sub-expressions for the application expression. We can and only can get the type when the first expression has a function type while the second expression has the type of the first parameter of that function.

$$\frac{\Gamma \vdash t : TBool \quad \Gamma \vdash thn : T \quad \Gamma \vdash els : T}{\Gamma \vdash EIf(t, thn, els) : T} \tag{6}$$

For if then else branches expression, we need to ensure the test expression has type bool, while both branches share the same type as the whole expression.

The above cases all do not include changing the context, while type relations for Lam and Rec, which both define a new function and introduce local variables, and add elements to the context.

$$\frac{\Gamma, x : T1 \vdash b : T2}{\Gamma \vdash EAbs(x : T1, b) : TFun(T1, T2)} \tag{7}$$

$$\frac{\Gamma, x : T1, f : TFun(T1, T2) \vdash b : T2}{\Gamma \vdash ERec(f, x : T1, T2, b) : TFun(T1, T2)} \tag{8}$$

The type relation for Rec is basically an extension of the Lam. It needs to check the declared return type is the same as the type of body with the parameter and the function itself with their types are added to the context.

```
1  | RecTyped (G : ctx) (f: string)
2      (x : string) (aa bb A: ty)
3      (e : exp)
4      (p1 : (ty.TFun aa bb) = A)
5      (p2: typed (ctx.ctx_snoc
6          (ctx.ctx_snoc G x aa)
7          f (ty.TFun aa bb)) e bb)
8      : typed G (exp.ERec f x aa bb e) A
```

Here is a code snippet for the type relation for recursive functions. A type A is introduced for all rules to represent the final type of the expression for all cases as shown in line 2, because it surprisingly simplifies the proof of the completeness of the type inference algorithm to only a $simp$ term for all cases there.

C. Some STLC with Recursive Examples

This section is for the exercise 3 in the project. Here we define a function that checks if two natural numbers are equal in STLC.

```
rec f(x : Nat) : Nat ->  Bool:
  return lambda y : Nat:
    if x == 0:
      if y == 0:
        return true
      else:
        return false
    else:
      if y == 0:
        return false
      else:
        return (f(x - 1))(y - 1)
```

Because the language we defined only supports lambda with one parameter, to compare two natural numbers, we need the outer function to get a nat and return a function from nat to bool.

In the inner function, we subtract both numbers by one and check if they reach zero at the same time.

The proof of: $\vdash f : TFun(TNat, TFun(TNat, TBool))$, is just a lot of $apply\ typed\_sth$ and $refl$. These $refl$s come because of those type As introduced in $typed$.

D. Type Inferencer for STLC with Recursive Function

The $typed$ inductive predicate defines in a certain context, what it requires for an expression to have a certain type, while it does not give an algorithm for checking whether the expression is typed or inferring its type.

Although the $typed$ type relation is not the type inferencer itself, it gives a syntactic directed type relation and is similar to the type inference algorithm, which means we can use a recursive function on all types of expressions to give an inferencer for STLC similar to the type relation rules. Take expression $ETrue$ as an example:

```
def type_infer : ctx -> exp -> option ty
| G exp.ETrue := some ty.TBool
```

Here a type of TBool is always returned when encountering a $True$ expression.

The type inferencer takes a context and an expression as input and an optional type as output. When the expression fails the type check in the context, a $none$ will be returned. In a real-world type inferencer, we might want to use $Pair(info, Option\ ty)$ as the return type to also maintain the error information.

The type checker really does something when it comes to some more complex expressions:

```
def type_infer : ctx -> exp -> option ty
| G (exp.EIf e1 e2 e3)      :=
  match type_infer G e1 with
  | some ty.TBool :=
    match type_infer G e2,
    type_infer G e3 with
    | some tA, some tB :=
        if tA = tB then (some tA)
        else none
    | _, _             := none
    end
  | _ := none
  end
```

When it comes to cases like the if expression, we need to recursively check if the condition expression is giving a bool type and if both branches give the same reasonable types. The checking of the equality of the type of the two branches shows the importance of the attribute of $@[derivedecidable\_eq]$ for $type$ type. Without that attribute, this whole function is no longer decidable and therefore illegal as a lean function.

E. Completeness of the STLC Type Inferencer

The completeness of the type inferencer means that for any correct expression in this context, the type inferencer can infer the correct type. Since the inductive predicate $typed$ is the typing rules definition in lean, the completeness of the type inferencer is:

$$\forall \Gamma \forall e \forall A.typed\ \Gamma\ e\ A \rightarrow type\_infer\ \Gamma\ e = some\ A \quad (9)$$

Since the type inference algorithm is fully based on the typing rules and the propositions given by $typed$ mostly contains everything inference need, the proof of completeness is straightforward by first introducing the $typed$ side then do induction on $typed$. For each type of expression, the proof of completeness is mostly $simp$ by definition of $type\_infer$ and other assumptions.

F. Soundness of the STLC Type Inferencer

As the reverse of completeness, the soundness of the type inferencer means that all inferred type and expressions in the context, you can prove its correctness with the type relation rules. The soundness of the type inferencer is:

$$\forall \Gamma \forall e \forall A.type\_infer\ \Gamma\ e = some\ A \rightarrow typed\ \Gamma\ e\ A \quad (10)$$

The proof of soundness is much harder than the proof of completeness. The first steps are straightforward, we introduce variables and hypothesis, then do induction on the expression and the $type\_infer$ algorithm.

When it comes to cases like $Lam(x : T, b)$, after simplify the hypothesis on $type\_infer$ with its definition, we first need to remember the type of the variable in the context using : $generalize\ h : x : T \in \Gamma = G$, and also the inference of subexpression: $generalize\ ht : type\_infer\ G\ e = TO$. [1] Otherwise, if we directly do induction on $type\_infer\ G\ e$, we might lose some important type information for the last few steps that prevent lean from deciding the metavariables for types.

Here, the lean tactic $finish$ can be used, as there are a lot of cases where we just induction on the expression and discard

all impossible cases using $by\_contradiction$ proof. However, I am not using them because it truly requires a lot f CPU resources. Since I prove them step by step, the proof becomes much longer.

III. System F

A. Definition of Abstract Syntax, Type and Context

For the extension part, I extended the previous part into the polymorphism of system F. Two new types $TVar$ for type variables and $\forall$ all type are added:

$$\begin{aligned} ty &= X \\ &= \forall X.ty \end{aligned} \tag{11}$$

here X is a type variable with name x, and $\forall X.ty$ is the means for all type X,ty.

Two new expressions are added also:

$$\begin{aligned} exp &= \lambda X.exp \\ &= exp[T] \end{aligned} \tag{12}$$

which means type abstraction to construct $forall$ type and fix $forall$ type to type T.

Here we need to find a good way to represent the type variables. And there are some common way for this [2]

1) Naming all type variables the same as we do for a variable in expression.
2) Naming all type variables with a unique name.
3) Fully redesign the language with combinators without any variables.
4) Using a representation without symbolic names.

Because of the type application expression, we will introduce a type substitude for variables, which makes the first two choices bad because it would be hard to implement a renaming and substitude mechanism for them. Even if all variables are assigned a different name, they are still not substitute stable. Thus we choose the third approach with de Bruijn index which use the index of the variable in the expression to represent it.

To keep it consistent, we use de Bruijn index for both type variables and term variables, thus the type becomes:

$$\begin{aligned} ty &= I : N \\ &= \forall.ty \end{aligned} \tag{13}$$

$$exp = i : N\ Var \tag{14}$$

Then we define a type binder:

$$\begin{aligned} binder &= Var(ty) \\ &= Type \end{aligned} \tag{15}$$

‘@[derive decidable_eq]‘ is also added to type binder to check equality for it. Then the context becomes just a list of binders. When a new type variable is added to the context, we simply append a type binder into the context with no real information, when a term variable is added to the context, we append its corresponding type to the context.

Since all variables become a natural number, we can look up the variable in the context with $love$ provided $list.nth$ to get an option binder and destruct to get the type.

B. Substitution and Typing Rules for System-F

Here we need to add typing rules for the two newly added expressions type abstraction and type application.

$$\frac{\Gamma, X \vdash t_2 : T_2}{\Gamma \vdash \lambda X.t_2 : \forall X.T_2} \tag{16}$$

The implementation of typing rules for type abstraction is trivial, we add a new type binder to the context and the type of body with a forall is its type.

```
| TAbs_typed {G : list bi} (e: exp) (A T: ty)
(p1: typed (add_bi G bi.TarBind) e A)
(p2: T = ty.TForAll A): typed G (exp.ETAbs e) T
```

For type application, we have the typing rule:

$$\frac{\Gamma \vdash t_1[T_2] : [X \to T_2]T_{12}}{\Gamma \vdash \lambda X.t_2 : \forall X.T_2} \tag{17}$$

this introduces a type substitution that we need to substitute X with T_2 in T_{12}.

To define a substitute, we first need an auxiliary function shift so that we can renumber the index of a free variable. When things like $[0 \to S]\lambda.1$ happens, free variables inside S should be shifted by 1.

```
def typeShiftAbove : ty -> N -> N -> ty
| (ty.TVar x) d cc:= if x >= cc then
ty.TVar (x + d) else ty.TVar(x)
| (ty.TFun t1 t2) d cc:= ty.TFun
(typeShiftAbove t1 d cc) (typeShiftAbove t2 d cc)
| (ty.TForAll t) d cc:=
ty.TForAll (typeShiftAbove t d (cc + 1))
```

Here d means how much we need to shift the variable and cc is a cutoff that controls which variables should be shifted.

Shift function should also support negative shift, which is decrease, since we do not have access to negative integers in Lean, we define a typeShiftBelow to do the negative shift. For most cases, we consider cutoff to be 0, so we define tsa and tsb for cc to be 0.

Thus we can define type substitute with type shift:

```
def typeSubst : ty -> ty -> N -> ty
| (ty.TVar i) tyS c :=
if i = c then tsa c tyS else (ty.TVar i)
| (ty.TFun t1 t2) tyS c :=
ty.TFun (typeSubst t1 tyS c) (typeSubst t2 tyS c)
| (ty.TForAll t) tyS c :=
ty.TForAll(typeSubst t tyS (c + 1))
| tyT _ _ := tyT
```

Consider $[X \to S]T$, here the first parameter is the X, the second is the S, c is still the cutoff.

For the case that we have in type application, we first type shift it by 1, then type substitute, the decrease it by 1, which is:

```
def typeSubstTop (tyS tyT : ty): ty :=
  tsb 1 (typeSubst tyT (tsa 1 tyS) 0)
```

Then we can define the typing rule for it:

```
| TApp_typed {G : list bi} (tt1 : exp)
(TT2 A : ty) {TT12 : ty}
(p1 : typed G tt1 (ty.TForAll TT12))
(p2 : typeSubstTop TT2 TT12 = A) :
typed G (exp.ETApp tt1 TT2) A
```

C. Type Inference for System F

The type infer function becomes simple after we have solid typing rules and enough auxiliary function for type substitute. We only need to extend it for the two newly added expression type abstraction and type application:

```
| G (exp.ETAbs e) :=
  match
  type_infer (add_bi G (bi.TarBind)) e
  with
  | some C := some (ty.TForAll C)
  | _ := none
  end
| G (exp.ETApp e t) :=
  match type_infer G e with
  | some ty1 := match ty1 with
    | ty.TForAll ttt :=
        some (typeSubstTop t ttt)
    | _ := none
    end
  | _ := none
  end
```

The proof of completeness is still simple even for them. For case type abstraction, a chain of simps are enough, for case type application, a chain of simps with a finish is enough.

The proof of soundness is similar to the previous ones.

The case type abstraction is very similar to lambda. The case type application is a little bit more complicated, but some finishes are used to make it a really short proof, but still similar to the case of lambda.

IV. Conclusion and Personal Thoughts

In this project, type judgment and inference for both simple type lambda calculus and system F are implemented. Most features in lean are pretty handy, auto deriving for equal, finish and other features are really useful. And I am happy that most functions I write in other functional programming languages can be rewritten in lean so easily. For example, I wrote the type inference first in Fsharp, and I change only a little for it to work in lean.

It is just sometimes I hope Lean can consider $A = B$ as $B = A$ as same thing. For now, If I have $h1 : B = A, h2 : C$ when I need to use B to substitute all A in C, I might to first get $h3 : A = B$ and use rw after that.

In short, Lean is strong enough for me, and this simple type inferencer, or maybe type checker. I would want to extend this to be a real type inferencer by implementing extension 5 or some subtyping.

References

[1] B. C. Pierce, C. Casinghino, M. Gaboardi, M. Greenberg, C. Hriţcu, V. Sjöberg, and B. Yorgey, "Software foundations," *Webpage: http://www.cis. upenn. edu/bcpierce/sf/current/index. html*, 2010.

[2] B. C. Pierce, *Types and programming languages*. MIT press, 2002.