Bidirectional Type Inference and Checking (Draft)

Isitha Subasinghe

26th November 2021 (Draft)

Abstract

Type theory is infamously considered difficult to get started in. There is simply too much to learn and this information is scattered all over the place. This paper addresses this issue by providing the basics for building a state of the art type inference and checking algorithm based on bidirectional type inference as described by Dunfield and Krishnaswami.

1 Introduction

Type inference is at the core of modern langauges, yet the knowledge to implement these algorithms are not easily accessible. Understanding type inference requires the reader to have a grasp of Type Theory, Set Theory, Lambda Calculus and First Order Logic, this paper aims to reduce the burden of learning type inference and provides a central location for a reader to understand enough of these concepts to get started on programming their own implementation of the previously mentioned type inference algorithm.

2 Related Work

Hindley-Milner type inference, discovered by Hindley and Milner independently have two known algorithms for type inference, Algorithm W and Algorithm J, both of which require a certain cognitive effort by the reader.

3 Setup

Let's define a module called "Eval" used to contain our algorithm. This is a simple but necessary step for any Haskell program.

$\{\#-LANGUAGE\ DataKinds, GADTs, KindSignatures, StandloneDeriving-\#\}$
module *Eval* **where**
import *qualified Data.Set as S*
import *qualified Data.Map as M*
import *Control.Applicative*
import *Data.Semigroup*
import *Data.Monoid*

In addition to the above code we need another module called "Main" to function as our entry point. We mostly don't have to worry about this Module, it is a simple interface to interact with the "Eval" module.

```
module Main where
```

4 Lambda Calculus

Knowledge of lambda calculus is required to understand some of the type theory needed to understand the paper referenced previously. Here we provide a quick, mediocre introduction to lambda calculus. What is lambda calculus you may ask, Michaelson states that λ calculus is a system for manipulating λ expressions. This is not entirely useful yet but it should make more sense when we explore what λ expressions are.

4.1 λ expressions

λ expressions are quite simple, they are either a name to identify an abstraction, a function or a function application to specialize an abstraction. The BNF for λ calculus is given below:

$$
\begin{aligned}
\langle \text{expression} \rangle &::= \langle \text{name} \rangle \mid \langle \text{function} \rangle \mid \langle \text{application} \rangle \\
\langle \text{name} \rangle &::= \texttt{a...z} \\
\langle \text{function} \rangle &::= \lambda \langle \text{name} \rangle . \langle \text{body} \rangle \\
\langle \text{body} \rangle &::= \langle \text{expression} \rangle \\
\langle \text{application} \rangle &::= (\langle \text{function expression} \rangle \ \langle \text{argument expression} \rangle) \\
\langle \text{function expression} \rangle &::= \langle \text{expression} \rangle \\
\langle \text{argument expression} \rangle &::= \langle \text{expression} \rangle
\end{aligned}
$$

4.2 Free Variables

Given an expression e, the following rules define $FV(E)$, the set of free variables in e:

- If e is a variable x, then $FV(e) = x$.
- If e is of the form $\lambda x.y$ then $FV(e) = FV(y) - \{x\}$.
- If e is of the form xy then, $FV(e) = FV(x) \cup FV(y)$.

4.3 Bound Variables

4.4 β reductions

β reductions simply define how we may remove a definition of a function application by moving the argument inside the function. The reduction rule is given below.

$$(\lambda x.e_1) \Rightarrow e_1[e_2/x]$$

Below are some examples to ground the reduction rule provided above.

- $(\lambda x.x)(\lambda y.y) \Rightarrow (\lambda yy)$
- $(\lambda x.xx)(\lambda y.yy) \Rightarrow (\lambda y.y)(\lambda y.y)$
- $(\lambda x.x(\lambda x.x))y \Rightarrow y(\lambda x.x)$

4.5 η conversions

η reductions allow you to convert between $\lambda x.(f\ x)$ and f whenever x is not a free variable in f. Two λ expressions ($\lambda x. + 1x$ and $(+1)$) are equivalent in the sense that these expressions behave in exactly the same way when they are applied to an argument – they add 1 to it. In general, if x does not occur in F, then $\lambda x.Fx$ is η-convertable to F.

5 Set Theory

6 First Order Logic

7 Type Theory

8 Simply Typed Lambda Calculus

Simply Typed Lambda Calculus can be given by the following grammar presented in BNF form.

$$
\begin{array}{rcl}
\langle t \rangle & ::= & \texttt{x} \ldots \texttt{z} \text{ – } \textit{Variables} \\
& & \mid \langle t\ t \rangle \text{ – } \textit{Application} \\
& & \mid \langle \lambda\ \mathrm{x}\ .\ t \rangle \text{ – } \textit{Abstraction} \\
& & \mid \textit{true} \mid \textit{false} \\
& & \mid \textbf{if}\ t\ \textbf{then}\ t\ \textbf{else}\ t \\
& & \mid t : \tau \\
\langle \tau \rangle & ::= & \langle \mathrm{Bool} \rangle \\
& & \mid \langle \tau \to \tau \rangle
\end{array}
$$

The data structures used for evaluating the presented lambda calculus is provided below. Note that the grammar has been slighly augmented to account for typing judgements, the constructor 'TypeAnnotation' is this augmentation.

```
data T
  = Var String
  | Application T T
  | Abstraction T T
  | TTrue  -- Avoid conflict with Haskell's 'True'
  | TFalse  -- Avoid conflict with Haskell's 'False'
  | IfElse T T T  -- If [Expr] Then [T1] Else [T2] where T1 == T2
  | TypeAnnotation T Type  -- Type annotation to mark that 'T' is of type 'Type'
  deriving (Show, Eq)
data Type
  = TBool
  | TApp Type Type
  deriving (Show, Eq)
```

Now we can finally start on type inference. First let's define our type signature for our inference function.

The above definition is quite simple, it defines a function that accepts a 'Context' ('M.Map String Type'), an expression ('T') and returns an algebraic data structure wrapping a type('Type'), this ADT is 'Maybe Type'. The usage of the 'Maybe' ADT is because the return value of the function could be nothing ('Nothing' is the function that constructs the Maybe ADT in this case) or something ('Just(x)' is the function that constructs the Maybe ADT in this case). This 'Context' corresponds directly to the Γ symbol seen in type theory literature.

Note It is important to note two symbols that are needed to understand literature on bidirectional type inference/checking.

- $\Rightarrow$ This is used in the form of x $\Rightarrow$ ty, it means "x is infered to be of type ty".
- $\Leftarrow$ This is used in the form of x $\Leftarrow$ ty, it means "x type checks against type ty".

Variables Let's define the rule for synthesis of a type of a variable.

$$\frac{(x : \tau) \in \Gamma}{\Gamma \vdash x \Rightarrow \tau}$$

This is quite simple to infer the type for, since the type annotation already exists in our context.

Booleans Another simple construct to type check are the boolean literals 'true' and false'. The rules for inference are given below.

$$\frac{}{\Gamma \vdash \textbf{true} \Rightarrow \textbf{Bool}}$$

$$\frac{}{\Gamma \vdash \textbf{false} \Rightarrow \textbf{Bool}}$$

The below code is all we need for dealing with boolean literals.

```
inferType :: M.Map String Type → T → Maybe Type
inferType ctx (Var s) = lookupTy s ctx
inferType ctx (TTrue) = Just TBool
inferType ctx (TFalse) = Just TBool
```

Annotation Now we examine how we can verify type annotated expression. The grammar for inference/checking is shown below.

$$\frac{\Gamma \vdash t \Leftarrow \tau}{\Gamma \vdash : \tau \Rightarrow}$$

```
inferType ctx (TypeAnnotation t ty) = checkType ctx t ty
```

'lookupTy' is given by the below code:

```
lookupTy :: String → M.Map String Type → Maybe Type
lookupTy x ctx = M.lookup x ctx
```

IfElse Here we observe how 'IfElse' may be checked.

$$\frac{\Gamma \vdash t_1 \Leftarrow \textbf{Bool} \qquad \Gamma \vdash t_2 \Leftarrow \tau \qquad \Gamma \vdash t_3 \Leftarrow \tau}{\Gamma \vdash \textbf{if } t_1 \textbf{ then } t_2 \textbf{ else } t_3 \Leftarrow \tau}$$

```
checkType :: M.Map String Type → T → Type → Maybe Type
checkType ctx (IfElse t1 t2 t3) ty =
  case (checkType ctx t1 TBool
    , checkType ctx t2 ty
    , checkType ctx t3 ty) of
      (Just bty, Just ty2, Just ty3) → Just ty
      _ → Nothing
```

Abstraction

$$\frac{\Gamma, x : \tau_1 \vdash t \Leftarrow \tau_2}{\Gamma \vdash \lambda\ x.\ t \Leftarrow \tau_1 \to \tau_2}$$

```
checkType ctx (Abstraction x t) ty12 =
  case ty12 of
    (TApp ty1 ty2) → checkType ctx t ty2
    _ → Nothing   -- invalid type since an Abstraction has to be of type TApp
```

Application

$$\frac{\Gamma \vdash t_1 \Rightarrow \tau_1 \to \tau_2 \quad \Gamma \vdash t_2 \Leftarrow \tau_1}{\Gamma \vdash t_1\, t_2 \Rightarrow \tau_2}$$

```
Check Inference   checkType ctx t ty = case inferType ctx t of
  Just ty' → if ty' ≡ ty then
    Just ty'
    else Nothing
  Nothing → Nothing
```

9 Complete and Easy Bidirectional Typechecking for Higher-Rank Polymorphism

I think we are now finally at a place where we can start on Dunfield and Krishnaswami's work. I understand that this is quite a lot to grasp, but I believe this is as small as I can keep this without threatening the goals of this paper.

9.1 Expressions

Let's define the expressions in our langauge.

```
data Expr
  = EVar Var
  | EUnit
  | EAbs Var Expr
  | EApp Expr Expr
  | EAnno Expr PolyType
  deriving (Show, Eq)
newtype Var = Var String deriving (Show, Eq, Ord)
data TypeKind = Mono | Poly
data Type' :: TypeKind → ∗where
  TUnit   :: Type' a
  TVar    :: TVar → Type' a
  TExists :: TVar → Type' a
```

A Appendix

A.1 Notation