How to play type inference

The original documentation can be found here: TypeInferencer Repository: https://github.com/kekyo/TypeInferencer, version 1.0.3.

This document is intended for the following readers:

- Want to try `TypeInferencer`.
- Those who are interested in type inference and type inferencers.
- First-time learners of type inference and type inferencers.

The following are not explained:

- A detailed description of the internals of type inferencers (Related articles and type theory. Also, the `TypeInferencer` is intended to be as plainly implemented as possible).

Please note that I am not familiar with it myself, so some parts may be incorrect. The purpose of this document is also to organize my understanding.

(This document was written in Japanese, then translated into English by machine translation and reworked. I'd be happy if you could help me out with any obvious grammar mistakes, or phrases that are difficult to understand, through PR, etc.)

Type Inference Overview

`TypeInferencer` is a plain and compact implementation of "type inference." In general, the type inferencer is implemented inside a language processor. The language processors we are most familiar with, such as `F#`, `OCaml`, `Haskell`, and `Java`, have static type systems. Alternatively, dynamic type systems may do some type inference to optimize their execution.

Type inference is the automatic computation and indication of the type of an expression by the processor, without explicitly indicating the type. For example:

```
// [*] F# expression - (1)
let f x = x + 1

// [*] Result of inference - (2)
let f (x:int) : (int -> int) = x:int +:(int -> int) 1:int
```

- Note: Marked with `[*]` to distinguish F# pseudo code from tries out `TypeInferencer`.

Usually, when people write code, they do not specify the type as in (1). Therefore, by using a type inferencer, it can automatically calculate and derive what type can be applied to each term of an expression, as in (2).

Although various methods have been devised for type inference, `TypeInferencer` implements two methods, `Algorithm W` and `Algorithm M`, which are typical methods for introducing type inference, and can be used in different ways.

- `Algorithm W` determines the type of an expression from the top (root) to the bottom (leaf) on the syntax tree.
- `Algorithm M` does the opposite, it determines the type of an expression from the bottom (leaf) to the top (root) on the syntax tree.

(A "syntax tree" is an abstract syntax tree or called `AST`. It will be shown later.)

There is no difference in the meaning of the inferred results in both cases. First, `Algorithm W` was presented, and later `Algorithm M` was presented. The reason why different methods were researched is that `Algorithm M` is better at localizing where the type problem is found.

Although `TypeInferencer` does not specify the exact location of the problem when it occurs during type inference, many language processors and IDEs may want to point out the error as locally as possible in the code. In such a case, `Algorithm M` can give a more desirable result.

What TypeInferencer can do

`TypeInferencer` is a type inferencer. Usually, a computer language processor contains the following elements:

1. `Lexer` and `Parser`: The source code is written in a text-based format. The first step is to reinterpret this into a structure that is easily recognized by the computer. The `Lexer` and `Parser` play this role and generate a syntax tree from

the source code.
2. `Type checker`: A type checker, which types each node of the syntax tree appropriately. Alternatively, it can check if it is properly typed or not.
3. `Reducer` or evaluator: Evaluates each node of a syntax tree. Evaluating results is same as execution of an expression. Typically corresponds to the "Interpreter".
4. `Code generator`: Generates independent code from a syntax tree. Generates code that has running directly on the target. This is generally equivalent to "Compiler".

Not all language processing systems implement all the features. For example, a processor based on a dynamic type system might omit steps 2 and 4.

We can say that `TypeInferencer` is a part of the implementation of `Type checker` in 2. Therefore, you need to prepare a syntax tree as input. This can be written as follows:

```
// Load NuGet package.
#r "nuget: TypeInferencer"

// Namespace and module to use
open TypeInferencer

// (boilerplate up to this point, omitted in below examples)

// ----------------------

// Assemble a syntax tree, equivalent to the following code fragment:
// `let f = fun x -> x 1 in f`
let expr =
    ELet("f",                               // let f =
        EAbs("x",                           // fun x ->
            EApp(EVar "x", ELit(LInt 1))    // x 1
        ),
        EVar "f"                            // in f
    )

// (After this, use TypeInferencer)
```

You may be wondering, "Isn't it possible to write expressions in a more natural way?":

```fsharp
// Write the source code as a string
let code = "let f = fun x -> x 1 in f"

// Can we convert it to a syntax tree?
let expr = parse code
```

It would be more convenient if you could write something like. The role of the `Lexer` and `Parser` is to do such conversions. Therefore, the `TypeInferencer` does not include such a function.

But there is no need to be discouraged. If you need to tackle such a task in real job, it is already in your hands. The F# Compiler Service is it.

It covers all the features of the complete F# language processor API set. It includes all of the public APIs for `Lexer`, `Parser`, `Type checker`, and `Code generator`, which can be used individually or in combination. You may even integrate them into your own applications.

However, the F# language specification is complex, and the syntax tree used to represent its structure is also huge and complex. For many people, the F# syntax tree is not easy to understand why such a node definition is necessary, and it is also difficult to handle.

On the other hand, `TypeInferencer` is focused only on type inference, and has a minimal number of syntax tree definitions. In other words:

- What is a syntax tree?
- Specific methods of type inference.
- What happens when we perform type inference, or how a particular syntax tree is inferred.

In particular, since F# 6.0, the debugging information has been greatly improved, so using the debugger to trace the internal implementation will help you understand it.

(In that case, you may want to clone this repository and run the debug on the test code instead of the NuGet package.)

Perhaps if you can understand the structure of a simply type system such as `TypeInferencer`, you can practice metaprogramming using more complex and real world language processors such as F# compiler services.

TypeInferencer syntax tree

The nodes of the syntax tree are defined recursively as follows:

```
// Type for each node in the syntax tree.
type public Exp =
    | EVar of name:string
    | ELit of literal:Lit
    | EApp of func:Exp * arg:Exp
    | EAbs of name:string * expr:Exp
    | ELet of name:string * expr:Exp * body:Exp
    | EFix of func:string * name:string * expr:Exp
```

There are six types of nodes in total. Each of them is described below.

ELit (Literal values)

Let's start by inferring the type of the literal value, which is the easiest and simplest to understand. Literal values in the code are represented by a node called `ELit`. The following defines an integer value of `123` and infers it:

```
// Literal value: 123
let expr = ELit (LInt 123)

// actual = TInt
let actual = infer TopDown (TypeEnv []) expr
printfn "%s" (show actual)
```

- `TopDown` is also fine with `BottomUp`. The difference is whether to use `Algorithm W` or `Algorithm M`. In `TypeInferencer`, the meaning of the result does not make any difference.
- `TypeEnv` is called the "type environment", but for now we'll assume it's code generated from an empty list.
- The `show` function converts the resulting type to a human-readable string (called "pretty printer"). For literal-valued types, you will get results without the `T` prefix, like `Int`. If you want to get the strict definition, use `printfn "%A" actual` instead.

If you apply this syntax tree to the `infer` function, you will get back a value of `TInt`. This means that we have inferred that the value `123` is an integer type.

Similarly:

```
// Literal value: true
let expr = ELit (LBool true)

// actual = TBool
let actual = infer TopDown (TypeEnv []) expr
printfn "%s" (show actual)
```

In this case, `TBool` is returned. We have literally inferred that it is a boolean type.

The definitions of the `Lit` types that can be specified as arguments to `ELit` are as follows:

```
// The types used to specify arguments to ELit.
type public Lit =
    | LInt of value:int32
    | LBool of value:bool
```

You may want to try other types of values, but see also the definitions of `TInt` and `TBool` before you do:

```
// A type that indicates the type of the syntax tree
// (see below for TVar and TFun)
type public Type =
    | TInt
    | TBool
    | TVar of name:string
    | TFun of parameterType:Type * resultType:Type
```

`TInt` and `TBool` are values defined with the type `Type`. (You may be confused about what is a “type” and what is a “value” until you get used to it. Also, the `Type` type is not the `System.Type` type in .NET, which is `TypeInferencer` own definition.)

Actually, there are only four types that `TypeInferencer` can recognize. In other words, for literal values, there are only integer types and boolean types. Therefore, to support other types (e.g., floating point, string, etc.), we need to increase the number of these variations or devise a better internal implementation.

There are other questions as well. Since we have gone to the trouble of defining the nodes of the parse tree as `LInt` and `LBool`, isn’t it natural that their types should be attributed to `TInt` and `TBool`? This is the point.

In fact, even the internal implementation returns the result by hardcoding it this way.

`TypeInferencer` did not increase the number of type definitions, nor did it introduce flexibility. Modifying it to add more types of literal values is relatively easy, but as shown earlier, it is confusing if you are not familiar with “value”, “type”, “value of type”, etc.

The current internal structure of `TypeInferencer` helps us understand it, thanks to F#’s guarantee of implementation consistency (literally, with F#’s type system). This is also because the goal was to reproduce as closely as possible the articles on which the implementation was based.

EAbs and EVar (Anonymous functions and free variables)

An anonymous function is an expression using `fun` keyword in F#:

```
// [*] Define an anonymous function and bind it with the name `f`.
let f = fun x -> x + 1
```

Define such an expression with `EAbs` (note that `let` is not included):

```
// Define an anonymous function.
// `fun x -> 123`
let expr = EAbs("x", ELit(LInt 123))

// actual = a0 -> TInt
let actual = infer TopDown (TypeEnv []) expr
printfn "%s" (show actual)
```

The result of evaluating this syntax tree is `a0 -> TInt`, and this type is defined by `TFun`. This means “a function type that takes an unknown type `a0` and returns type `TInt`”.

Since the body expression of `EAbs` is `LInt`, we know that it returns `TInt` in a fixed way. On the other hand, there is no information about the parameter variable `x`, and it is not used in the body expression, so there is nothing knowledge for inference. In the finishing, the inference ends with the type of `x` unknown, and the parameter argument part is of type `a0`.

The unknown type `a0` acts as a placeholder, and if the type is found during inference, it will converge to that type. This placeholder is called a “type variable” (or “free type variable”) and is represented by `TVar`.

The function type `TFun` and the type variable `TVar` can also be defined manually as follows:

```
// a0 -> TInt
let typ = TFun(TVar "a0", TInt)

// Make sure it matches exactly - (1)
System.Diagnostics.Debug.Assert((typ = actual))
```

To check the content of a type, you can use a pattern matching expression:

```
// Make sure it matches exactly - (2)
System.Diagnostics.Debug.Assert(
    match actual with
    | TFun(TVar name, TInt) when name = "a0" -> true
    | _ -> false
)
```

`TypeInferencer` assigns a automatic generated number to the placeholder. Note that this number may vary depending on the structure of the syntax tree. Therefore, compares rather than defining it manually, it is recommended to use it to check if it is consistent with other inferred `TVar`.

Here is also a description of `EVar`, which makes it possible to refer to the parameter argument variable `x` (not `TVar`):

```
// Define an anonymous function and use the parameter variable
// in the body expression.
// `fun x -> x`.
let funcexpr = EAbs("x", EVar "x")

// actual = a0 -> a0
let actual = infer TopDown (TypeEnv []) funcexpr
printfn "%s" (show actual)
```

You can specify the variable you want to refer to with `EVar`, which is called a “free variable”. (Parameter variables are sometimes called “bound variables” because they can be considered bound.)

Although `EVar` and `TVar` are similar, the difference is whether the target is a concrete value or a type.

The result `a0 -> a0` shows that “the whole thing is a function type, and the argument and result types are both unknown, but they match.

It would not be very interesting to just define an anonymous function. We need to “apply” the following function.

EApp (Function application)

Function application means to define a function and apply it to the arguments as follows:

```
// [*] Define an anonymous function in F#.
let f = fun x -> x
// let f x = x      // Can also be defined as `let` function in F#.

// [*] Apply function to argument (result is 123)
f 123
```

`EApp` is a node that defines such a function application:

```
// Define an anonymous function.
//  `fun x -> x`
let funcexpr = EAbs("x", EVar "x")

// Apply anonymous function to 123
//  `(fun x -> x) 123`
let expr = EApp(funcexpr, ELit(LInt 123))

// actual = TInt
let actual = infer TopDown (TypeEnv []) expr
printfn "%s" (show actual)
```

This brings us to an example that we can call “inference.” Since the anonymous function returns the parameter argument `x` as it is, we should be able to infer that it returns the same type as `x`. Therefore, we can infer that the type `TInt` of `123` given by the function application is returned by the anonymous function, resulting in `TInt`. Similarly:

```
// Apply the anonymous function to true.
//  `(fun x -> x) true`
let expr = EApp(funcexpr, ELit(LBool true))

// actual = TBool
let actual = infer TopDown (TypeEnv []) expr
printfn "%s" (show actual)
```

Although it is the same definition of an anonymous function, the inferred result changes depending on the type of the argument given, and we can confirm that it is indeed inferred.

Now let’s look at some more different examples:

```
// Apply function true to 123 (?)
// `true 123`
let expr = EApp(ELit(LBool true), ELit(LInt 123))

// uncaught UnificationException
let actual = infer TopDown (TypeEnv []) expr
```

The expression `true 123` is unsettling to look at, and at first glance we can tell that it is not going to work, but let’s try to figure out why:

- `true` is a literal value. Its type is boolean, or `TBool`.
- By applying `true` to `123`, we assume that `true` is a function. The type of a function is a “function type”, `TFun`, as explained in `EAbs`.

If we consider what the assumed function type is, we can think of it as “a function type that takes an integer type argument and whose result type is unknown.” In other words, it should be something like `TInt -> a0`.

And there is no way to reconcile a boolean type with a function type `TInt -> a0`. This problem was discovered in `TypeInferencer` during the process called “unification”, and an exception was thrown.

Coffee break

Here’s an interesting example of an anonymous function definition:

```
// `fun x = x x`
let expr = EAbs("x", EApp(EVar "x", EVar "x"))

// uncaught OccurrenceException
let actual = infer TopDown (TypeEnv []) expr
```

This expression will raise an exception. In fact, we can’t infer this syntax tree. Try to figure out why. You may not be able to sleep tonight…

ELet (Binding)

This node is equivalent to `let` expression, which has been shown in some examples of F# code. Let's review it just in case:

```
// [*] Binding an integer value.
let x = 123

// [*] Refer to the bound variable.
printfn "%d" x
```

If you have recently started using F#, you may not be aware of below. This notation is called the "lightweight syntax", and can actually be written as follows, which is semantically more accurate (called the "verbose syntax"):

```
// [*] Binding an integer value.
let x = 123
in
  // [*] Refer to the bound variable. (in the explicitly scope)
  printfn "%d" x
```

`in` keyword determines the scope of the binding, and the variable `x` can only be used within this scope. `ELet` defines this faithfully:

```
// `let x = 123 in x`
let expr =
    ELet("x",
        ELit(LInt 123),
        EVar "x"    // Expression in scoped body.
    )

// actual = TInt
let actual = infer TopDown (TypeEnv []) expr
printfn "%s" (show actual)
```

This syntax tree will eventually return `123`, so the inference result is `TInt`.

Of course, we can also bind anonymous functions:

```
// `let f = fun x -> x in f 123`
let expr =
    ELet("f",
        EAbs("x", EVar "x"),
        EApp(EVar "f", ELit(LInt 123))
    )

// actual = TInt
let actual = infer TopDown (TypeEnv []) expr
printfn "%s" (show actual)
```

At this point, it's already getting difficult for humans to infer types accurately. Here are some more interesting examples:

```
// `let f = fun x -> fun y -> x y in f (fun z -> z) 123`
let expr =
    ELet("f",
        EAbs("x", EAbs("y", EApp(EVar "x", EVar "y"))),
        EApp(EApp(EVar "f", EAbs("z", EVar "z")), ELit(LInt 123))
    )

// actual = TInt
let actual = infer TopDown (TypeEnv []) expr
printfn "%s" (show actual)
```

You can see that the type of the function, the result type of applying the function, and so on are correctly tracked. In this example, `x` is evaluated as a function type, which is interesting.

If you look at a complex syntax tree like this, you will notice something. Take a good look at the following code example:

```
// [*] Binding expression.
let x = true in 123

// [*] Anonymous function and function application.
(fun x -> 123) true
```

I should have understood this implicitly, but I was surprised when I actually saw the inferred result. In other words, the bound expression can be substituted by a pair of anonymous functions and function application. I will try to verify this on `TypeInferencer`:

```
// `let x = true in 123`
let expr1 = ELet("x", ELit(LBool true), ELit(LInt 123))

// `(fun x -> 123) true`
let expr2 = EApp(EAbs("x", ELit(LInt 123)), ELit(LBool true))

// actual1 = actual2 = TInt
let actual1 = infer TopDown (TypeEnv []) expr1
let actual2 = infer TopDown (TypeEnv []) expr2
printfn "%s = %s" (show actual1) (show actual2)
```

- Side note: In Javascript, the scope of variables is quite loose, and there is a danger of accidentally reassigning values. In such cases, there is a technique to create a pseudo-scope using anonymous functions and function application, as shown here.

Of course, nesting of `ELet` is also possible, although it is not shown here. This is something you should check out for yourself.

EFix (Fixed point)

In general, recursive functions cannot be defined simply by using `let`:

```
// [*] Define a recursive function.
let f x = f x // `f` is not defined
in
  f 123
```

In F#, you can define recursive functions by using `let rec` keyword:

```
// [*] Correct definition.
let rec f x: int = f x   // the result type cannot be inferred,
in                       // so it is assumed `int` (type annotation).
  f 123
```

- Note: Of course, this expression will fall into an infinite loop on F# environment.

The reason why this cannot be achieved with `let` is that the scope of free variables that appear in the expression to be bound is outside of `let` binding expression. As shown in `ELet`, the scopes that can refer to bound variables are after `in` keyword on `let ... in ...`. F#'s `let rec` relaxes this restriction.

In `TypeInferencer`, the use of `EFix` makes it possible to define recursive functions:

```
// `fix f x = f x`
let expr =
    EFix("f",
        "x",
        EApp(EVar "f", EVar "x"))

// actual = a0 -> a1
let actual = infer TopDown (TypeEnv []) expr
printfn "%s" (show actual)
```

In fact, this just defines a "recursive function" (and its type is `a0 -> a1`). To make it the same expression as `let rec`, bind it as follows:

```
// `let f = fix fi x = fi x in f 123`
let expr =
    ELet("f",
        EFix("fi",   // The name is changed to `fi` for clarity,
            "x",     // it is optional.
            EApp(EVar "fi", EVar "x")),
        EApp(EVar "f", ELit(LInt 123)))

// actual = a0
let actual = infer TopDown (TypeEnv []) expr
printfn "%s" (show actual)
```

Just as the result type could not be inferred in the F# code example, the `TypeInferencer` could not be inferred either. If you return the value of the result, you can infer the type of the value as a clue.

Note that the name `EFix` comes from "fixed point" or "fixed point combinator". The concept of fixed point is hard to understand intuitively, so it is better to refer to various references. (Example from Wikipedia)

Limitations and evolutionary topics

Above is a description of the syntax tree and type inferencing that `TypeInferencer` has. As partly shown in the text, it has the following limitations and their solutions compared to real world language processing systems:

- Literal values are only available in integer and boolean types.
    - It is not very difficult to change `ELit` to directly handle immediately values (all instances inherited from `obj` type) on the hosted runtime environment.
- There are not enough node types to perform practical calculations.
    - For example, lack for pattern matching expressions and `if ... then ... else`, we cannot define realistic expressions. In the description of `EFix`, we could not show a realistic example of a recursive function because we could not implement a stop condition.
    - It is possible to define a node type like `EIf`, and it has been well researched, so you should follow various articles.
    - It may not be necessary to mention it again, but syntax trees similar to `for` and `while` that constitute loop logic can be defined by applying with `EFix`, since we can construct by recursive functions. Of course, it is also a good idea to try to define node types that correspond directly to these. In that case, we need to face a fundamental question of functional languages: "whether a function returns a value or not."
- Type annotations cannot be applied to each node of a syntax tree.
    - As shown in the F# example in `EFix`, it is sometimes better to manually annotate some terms with types to make better inferences and help our understanding. You may also want to refer to the existing article on partial type annotation of syntax trees.
- Type composition is not possible.
    - Functional languages such as F# can define "tuple" and "algebraic data" types. It allow for more flexible and complex definitions, but `TypeInferencer` does not have such node type and inference capabilities.
    - Dealing with algebraic data types is complicated by how to handle the structure of records and whether to allow partial field alignment.
- Partial type is not possible.
    - Partial type means that the inheritance and implementation relationships of types in OOP type system allow type compatibility to more abstract types.
    - As we have already seen, `TypeInferencer` cannot define a type that is compatible with another type, because it only has the unknown type (`TVar`), `TInt`, `TBool`, and `TFun`. These can be achieved by calculating "type constraints" on the unknown types to see if they are compatible or not.
- The inference result will always return only the type for the whole expression.
    - In the IDE, you need to use the inference results for the nodes of the partial syntax tree to suggest the types corresponding to the context positions, but `TypeInferencer` discards the types obtained in partial inferencing.

    - This problem could be solved by storing the inference results for each node.

It would be a good idea to modify and add these from the `TypeInferencer` implementation. As a book that comprehensively covers the above, “Types and Programming Languages” is famous, but however, the level of difficulty is quite high, so it is not suitable for beginners. It is better to check the basics with books on “logic” and “symbolic logic” first before engage in.