Class Notes on Type Inference

2024 edition
Chuck Liang
Hofstra University Computer Science

Background and Introduction

Many modern programming languages that are designed for applications programming impose typing disciplines on the construction of programs. In constrast to untyped languages such as Scheme/Perl/Python/JS, etc, and weakly typed languages such as C, a strongly typed language (C#/Java, Ada, F#, etc ...) place constraints on how programs can be written. A type system ensures that programs observe logical structure. The origins of type theory stretches back to the early twentieth century in the work of Bertrand Russell and Alfred N. Whitehead and their *"Principia Mathematica."* They observed that our language, if not restrained, can lead to unsolvable paradoxes.

Specifically, let's say a mathematician defined S to be *the set of all sets that do not contain themselves.* That is:

$$S = \{all\ A\ :\ A \notin A\}$$

Then it is valid to ask the question *does S contain itself* ($S \in S$?). If the answer is yes, then by definition of S, S is one of the 'A's, and thus $S \notin S$. But if $S \notin S$, then S is one of those sets that do not contain themselves, and so it must be that $S \in S$!

This observation is known as *Russell's Paradox.* In order to avoid this paradox, the language of mathematics (or any language for that matter) must be *constrained* so that the set S cannot be defined. This is one of many discoveries that resulted from the careful study of language, logic and meaning that formed the foundation of twentieth century analytical philosophy and abstract mathematics, and also that of computer science. The type theory used in computer science today is derived from a 1940 paper by Alonzo Church called *"A Formulation of the Simple Theory of Types."* It was given in the context of Church's λ-calculus.

The Syntax of Types

Programming languages have a variety of syntax for expressing types. In general, type expressions are categorized into atomic or primitive types such as `int`, `double`, `char`, etc..., and compound types that involves a *type constructor.* Examples of type constructors include structs and classes. A struct (or class) containing an int and a char can be represented as $int * char$ type (a cartesian product). Additional type constructors depends on the language, such as * in C/C++ that indicates a pointer type. Functions have types of the form $A \to B$ where A represents the type of its parameters (domain) and B represents the return type (codomain). For example, a C function

```
int f(int x, char y)
```

can be said to have type $(int * char) \rightarrow int$. The $\rightarrow$ constructor associates to the right. `void` is the empty set and can be given the label 0 (in F#, it's called 'unit'). The void type should be associated with all formulas that generate side effects as opposed to returning a value. For example, a `cout` statement in C++ can be said to have the void type.

We write $S : \tau$ if expression S has type τ. Such formulas are called *type judgements.*

Type Checking Expressions

To say that a program is "well-typed" is to be able to assign a type to every subexpression of the program. A statement such as `cout` is well-typed if every expression processed by the statement is well-typed. However, just because a program is well-typed does not mean it's correct, only that it is logically structured. Each typed language defines a set of rules for determining what (if any) types each expression should have. We can study type theory in a pure setting using the *typed* λ-calculus. The λ-calculus contains two basic forms of expressions: applications of the form $(A\ B)$ and abstractions of the form $\lambda x.A$. Types for constants are assumed: for example, $3 : int$ is assumed to always be a true type judgement. Built-in functions are also considered constants: for example, '+' has type $(int * int) \rightarrow int$. Whether a function such as '+' is infix (in C) or prefix (in Scheme) is a purely syntactic matter and is irrelevant here: what matters is that it's a function that takes two integers and returns an integer[1].

Types for variables depend on a *type environment,* which is a set of type judgements of the form $x : \tau$. The type environment is usually labeled Γ. For example, if we make the following declarations

```
int x, y;
char z;
double u;
```

Then Γ will include $x : int,\ y : int,\ z : char,\ u : double$ for this section of the code. The set Γ is referred to as a *signature.* Java/C# interfaces and C/C++ header files are manifestations of the signature concept, but the idea of a signature is much more general than either of these constructs.

Given a signature, type judgements are inferred using one of the following rules:

$$\frac{S : \tau \in \Gamma}{S : \tau}\ id$$

[1]Of course, '+' is an overloaded operator that can also be used to add floats - which '+' we're talking about are often inferred from context.

$$\frac{A : \sigma \to \tau \quad B : \sigma}{(A\ B) : \tau}\ app$$

$$\frac{A : \tau \quad \text{with } x : \sigma \text{ temporarily added to } \Gamma}{\lambda x.A : \sigma \to \tau}\ abs$$

These rules can be read in both a forward and a backward ("goal-directed") manner. The first rule (id) simply says that if Γ already contains a judgement, then it's trivial to infer that judgement. For example, $x : int$ can be inferred trivially from the C-style declarations earlier. The second rule says that when you apply a function to an argument, the type of the argument must match the type of the domain of the function, and that the result of the application will be the type of the codomain of the function. The third rule is most interesting: it says that to show that a function has type $\sigma \to \tau$, *temporarily* assume that its parameter has type σ, and under this temporary assumption show that the body or return value of the function has type τ. Why temporarily? Because, as you should know well by now, formal parameters (λ-bound variables) have LOCAL scope. We must discard the assumption after applying this type inference rule so that it does not interfere with other parts of the program. If you were writing a compiler or interpreter, Γ is your "symbol table."

Product Types and Currying

When dealing with a cartesian-product type (for functions that take more than one paramenters, for example), we need the following rules:

$$\frac{A : \sigma \quad B : \tau}{(A, B) : \sigma * \tau}$$

$$\frac{(A, B) : \sigma * \tau}{A : \sigma} \qquad \frac{(A, B) : \sigma * \tau}{B : \tau}$$

These rules will combine multiple parameters into one parameter. It also allows us to use structs (or classes) that combine multiple elements into one object.

It is also always possible to convert a function that takes two (or more) parameters into a function that takes one parameter using a process called "Currying", after the logician Haskell Curry. That is, a function of type $A*B \to C$ can be converted to a function of type $A \to (B \to C)$. The second function takes the first parameter and *returns a function* that takes the second parameter. The technique also allows us to simplify our theory to deal only with functions that take one parameter. The language Ruby contains a feature that creates the curried version of a function; that is, if a function that takes two parameters is given only one actual parameter, a *function* that expects the other parameter will be returned. Currying is also used in code optimization. We can also implement currying fairly easily in langauges that allow the free manipulation of functions as lambda terms:

```
// In C#
using System;
public class curry
{
  public static Func<A,Func<B,C>> Curry<A,B,C>(Func<A,B,C> f)
  {     return x => y => f(x,y);   }
  public static Func<A,B,C> Uncurry<A,B,C>(Func<A,Func<B,C>> g)
  {    return (x,y) => g(x)(y);    }

  public static int h(int x, int y) { return x+y; } // test functions
  public static Func<int,int> g(int x) { return y=>x*y; }
  public static void Main()
  {
    Func<int,Func<int,int>> ch = Curry<int,int,int>(h);
    Console.WriteLine( ch(2)(3) );
    Console.WriteLine( Uncurry<int,int,int>(g)(4,5) );
  }//main
}
```

```
// in F#:
let Curry f = fun x -> fun y -> f(x,y);
let Uncurry g = fun (x,y) -> g x y;

let h(x:int,y:int) = x+y
let g x y = x*y
printfn "%d" (Curry h 2 3)
printfn "%d" (Uncurry g (4,5));;
```

Application

The type inference rules above can be used in an algorithm to determine if a program is well typed. For example, let Γ be the type signature $\{x : int,\ y : int,\ z : char,\ u : double,\ atoi : char \to int\}$ We can infer that $x + atoi(z)$ has type int using these rules[2]

$$\dfrac{\dfrac{x : int\ (\in \Gamma)}{(x, atoi(z)) : int * int} \quad \dfrac{\dfrac{z : char\ (\in \Gamma) \quad atoi : char \to int}{atoi(z) : int}\ app \qquad + : int * int \to int}{x + atoi(z) : int}\ app$$

Note that if we had $x + atoi(u)$ the inference above would fail because the *app* rule cannot

[2] `atoi` is a C library function that converts chars to their ascii values.

be applied to $atoi(u)$, since *double* does not match the domain type of *atoi* (char). Also note how we used the product rules to combine the two arguments of + into one.

Now let's infer that the function f below

```
int g(int x, char y) { return x + atoi(y); }  // assume well-typed
int f(int x)
{
  return g(x,'a');
}
```

is well-typed (that is, f has the type as declared). Assume it's already been established that function g has type $int * char \rightarrow int$.

$$\dfrac{\dfrac{x : int\ (assumed) \quad 'a' : char\ (constant)}{(x,' a') : int * char} \qquad g : int * char \rightarrow int\ (previous\ inference)}{\dfrac{g(x,' a'),: int\ (assume\ x : int)}{f : int \rightarrow int}\ abs}\ app$$

The Curry-Howard Isomorphism

When we type check a program, we making sure that the program's structure is logical. We're not using the word *logical* gratuitously here. Type theory holds a close relationship with formal mathematical logic. If we equate $\rightarrow$ with logical implication, product $*$ with conjunction (and), and every atomic type (int, char, etc) as *true,* then every type expression corresponds to a logical formula. Applying the type checking rules above under a signature Γ then corresponds to deducing that something is true under a set of logical assumptions. This important correspondance is called the *Curry-Howard Isomorphism.* It formally equates the process of writing programs with the process of formulating mathematical proofs. A particular consequence of this isomorphism is that *the type of every typable pure lambda term (combinator) is a propositional tautology.*

Unfortunately, not every lambda term is typable. The untyped lambda calculus is powerful enough to express all computations, but only a subset of computations are considered "type safe." In particular, the "bad" lambda term that has no normal form when applied to itself, namely $(\lambda x.x\ x)$, is not typable. To see why, apply the type inference rules above to try and derive a general type. We first associate some type variable σ temporarily with x. But in applying the typing rule for applications (labeled *app*), we see that in order for $x\ x$ to be typable, x must have type σ *and* $\sigma \rightarrow \tau$ for some type τ. These types are not compatible. In fact, in pure typed lambda calculus every term is guaranteed to have a normal form. You can't have infinite loops! We cannot derive the fixed-point combinator for recursion - we must import it as an addition to the calculus.

Typing Rules for Other Language Constructs

The rules above form the foundation of any type system. Specific langauges will require additional rules corresponding to the constructs that are in that language. For example, to type an assignment statement (`x=A;`) you will need to introduce an inference rule that checks that the left and right hand sides of `=` have the same type. Typing pointer expressions in C/C++ will require the following rules:

$$\frac{L : \tau}{\&L : \tau*} \qquad\qquad \frac{A : \tau*}{*A : \tau}$$

The first rule, however, does not distinguish L as an *L-value,* which is required before we place & before it. Only L-values have well-defined memory addresses. The typing rules must be used with additional restrictions of any given language.

Constructs such as nested {}'s in C can be modeled directly using λ-terms, and can be typed directly.

Conditionals such as if-else are interesting. We need to verify the types of three elements: the boolean and the types of both options. In many languages, the options for if-else are statements of type `void`. However, we should really consider the type of `return` construct as the type of the value that it returns. To make matters consistent, we should not think of if-else constructs as having type void, but rather the type of the values that it could return.

$$\frac{A : \tau}{return\ A : \tau} \qquad\qquad \frac{A : bool \quad B : \tau \quad C : \tau}{if\ (A)\ B\ else\ C : \tau}$$

In any typed language, you cannot have expressions such as

```
if (A) return "abc"; else return 2;
```

although such expressions are legal (and sometimes useful) in untyped languages such as Python. The down side of the flexibility of these languages is that it becomes impossible for the compiler to detect type errors before runtime.

Typing Recursive Functions

Recursive functions are manifestations of mathematical induction in computing. In type checking recursive code (tail or otherwise), we are essentially doing an inductive proof. Consider the factorial function:

```
unsigned int f(unsigned int n)
  { if (n==1) return 1; else return n*f(n-1); }
```

To prove that this function is well-typed for all positive integers, we first show that $f(1)$ is well typed (the base case), and under the *inductive hypothesis,* that $f(n-1)$ is well-typed:

$$\frac{\dfrac{n : uint\ (\in \Gamma) \quad f(n-1) : uint\ (ind.\ hypothesis)}{(n, f(n-1)) : uint * uint} \qquad * : uint * uint \to uint}{n * f(n-1) : uint}\ app$$

Homework Assignment (optional)

Using the typing rules described here, and following the examples shown (both here and in class), show that the following program fragments are well (or not well) typed:

1.

```
int f(double x, char y);  // assume well-typed
...

int x;
char y;
double z;
...
cout << (x+1) + f(z,y);   // type check this cout'ed expression.
```

That is, under a type environment Γ containing the declared types for f, x, y, z, show that the expression in the cout statement is well-typed.

2.

```
int g(int x, int y)
{
  return 3 + (y*x);
}
```

Please pay careful attention to how the rules should be applied and write out everything clearly and carefully. The type inference rules may seem tedious but they have to be in order to form the basis of an implementation, so observe them carefully. Be mechanical.

Subtyping and Parametric Types

"Subtype" is the word used to describe inheritance in type-theoretic research. A subclass is a subtype of the superclass. Types are related to sets in mathematics. However, when

we speak of a "class" we are really talking about the *description* of a set, not the set itself. Thus the subclass is seen as larger than the superclass, but in fact it describes a tighter set. Types such as *int* can also be considered a subtype of *double*, because there's a well-defined injective function from ints to doubles. If a term A is of type σ, which is a subtype of τ, then A is also of type τ.

A parametric type is a type expressions with variables, such ¡A¿ in C#or 'a in F#. However, **the rules of type checking described above can also be used to check the consistency of parametrically typed code.** Because of the relationship between logic and type theory, type expressions with variables are often written as being (second order) universally quantified with $\forall$. For example, if a function is of type $\forall a(a \to a)$, then the function can accept a value of any type as a parameter, as long as the value it returns is of the same type. If a function f has type $\forall a(a \to a)$ but we write:

```
int x = f("abc");
```

then we know this code is not typable *at compile time,* because the single type variable a cannot be instantiated with both `int` and `string` at the same time.

Type Inference

Just because a language has the syntax of types (C++) doesn't mean it has a meaningful type system. On the other hand, some languages may appear to not require types in its syntax, but in fact the opposite is true. In F#, types are seldom used directly by programmers. Instead, they're inferred from context. For example, the (tail-recursive) factorial function

```
let rec fact = function
  | (0,ax) -> ax
  | (n,ax) when n>0 -> fact(n-1,n*ax)
```

ML and F# allow definitions of functions by cases, similar to Prolog predicates and to a certain extent, the Visitor pattern in oop languages. But what is even more relevant is the absence of typing keywords in the code (i.e, "int"). ML *infers* that the type of this function is $int * int \to int$. From the type of 1 and $*$, it infers that the parameters and the return value must all be integers. The algorithm used for type inference is similar to the unification algorithm used in Prolog and AI. Furthermore, ML is able to infer *Hindley-Milner type schemes,* which are types with generic type variables. The significance of this ability is profound. It means that ML can automatically recognize the *naturally polymorphic* characteristics of programs. For example, one can define the length of a linked list in F# as follows:

```
let rec length = function
```

```
  | [] -> 0
  | head::tail -> 1 + length(tail);
```

F# is able to infer that the type of this function is `'a list -> int` where `'a` is a general type variable. This means that the function can be applied to linked lists containing data of any type. In stark contrast, the C++ type system is not powerful enough to even allow the *expression* of natural polymorphism, let alone infer it. Templates are not statically type checked. Instead, a new copy of the code is made for each new instantiation of the type variables, regardless of whether a new piece of code is really necessary (such as in the case of the length function). Java is a little better, but you have to write out the types yourself: it can't infer them.

ML is the language with which most of the applications of type theory was researched. One can think of ML as a typed version of Scheme: it is based on the *typed* lambda calculus. Slowly but surely, as technology becomes available to support the overhead of advanced language features, the research done with ML is finding its way into more popular, conventional languages. Generics were added to Java and C# around 2005, and now C# has increased type inference capabilities.

Exercise:

a. Using the rules of logical inference, derive the most general type (aka principal type scheme) of the S combinator, lambda x.lambda y.lambda z.(x z)(y z)

Check your answer with F#:

```
let S = fun x y z -> x z (y z):
```

b. Verify that the type of S is a propositional tautology by constructing a truth table.