Class Notes on Type Inference Calculi

Chuck Liang
Hofstra University Computer Science

Background and Introduction

Most modern programming languages designed for applications programming imposes a typing discipline on the syntax of programs. A type system ensures that programs observe a logical structure. The origins of type theory stretches to the early twentieth century in the work of Bertrand Russell and A. N. Whitehead and their "Principia Mathematica." The type theory used in computer science today is derived directly from a 1940's paper by Alonzo Church called "A Formulation of the Simple Theory of Types."

The Syntax of Types

Different programming languages will have different syntax for expressing types. In general, type expressions are categorized into "atomic" types such as `int`, `double`, `char`, etc... Compound types consist of a type constructor. For example, a struct or record containing an int and a char can be called *intXchar* type (a cartesian product). Different languages will have additional type constructors - such as * in C/C++ that indicates a pointer type. Functions have types of the form $A \rightarrow B$ where A represents the type of its parameters and B represents the return type. For example, a C function

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

can be said to have type $(intXchar) \rightarrow int$. The $\rightarrow$ operator associates to the right. `void` is the empty set and is given the label 0. 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 expression. A statement such as `cout` is well-typed if every expression processed by the statement is well-typed. Notice that just because a program is well-typed does not mean it's correct, only that it is logically structured. We know from the halting problem that certain properties of programs can never be pre-determined statically. A type discipline is just an attempt to induce a disciplined style of programming, and to catch certain common mistakes that arise in software.

Each typed language defines a set of rules for determining what (if any) types each expression should have. All such languages are based on 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 functions: for example, '+' has type $(intXint) \rightarrow int$. 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, in C/C++/Java 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.

Given an environment, type judgements are inferred using one of the following rules:

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

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

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

These rules can be read in both a forward and a backward ("goal-directed") manner. The first rules just 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 Java 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

[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.

the codomain (range) 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 that lies outside of the scope of the λ-binding. In other words, Γ here is our RUNTIME STACK!!!

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:

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

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

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". That is, a function of type $AXB \to C$ 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. It is also a technique used in code optimization.

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 environment $\{x : int,\ y : int,\ z : char,\ u : double\}$ We can infer that $x + atoi(z)$ has type int using these rules[2]

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

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

Note that if we had $x + atoi(u)$ the inference above would fail because the app rule can not 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 to + 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 right type). Note that f is just $\lambda x.(g\ x\ 'a')$. Assume it's already been established that function g has type $intXchar \to int$.

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

The Curry-Howard Isomorphism

Type theory holds a close relationship with mathematical logic. In fact, if we equate $\to$ with logical implication, product X with conjunction (and), and every atomic type (int, char, etc) as truth, then every type expression corresponds to a logical formula. Applying the type checking rules above under an environment Γ then corresponds to deducing that something is true under a set of logical assumptions. The important correspondance is called the *Curry-Howard Isomorphism.* It formally equates the process of writing programs with the process of formulating mathematical proofs (no kidding!).

Typing Rules for Language Constructs

The rules above form the foundation of any type system. Specific langauges will of course require additional rules, depending on 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:

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

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

Homework Assignment

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 already
...

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.

Assignment due Monday 3/19