CSC/MAT-220: Discrete Structures

Lab 6: ML

Due November 3, 2017

Type Inference. Up to this point, our programs have been written in *explicitly typed style*. That is, whenever we have introduced a variable, we have assigned it a type; i.e. every variable in a pattern has a type associated with it. A particularly pleasant feature of ML is that it allows you to omit the type information whenever it can be determined from context. For example, there is no need to give a type to the variable s in the function expression

$$fn\ s{:}string\ =>\ s\ \hat{}\ \text{“ ”}$$

since we are using the string concatenation operator. Therefore, we may write this expression as

$$fn\ s\ =>\ s\ \hat{}\ \text{“ ”}$$

allowing ML to insert ":string" for us. This is called the *principle typing property* of ML: whenever type information is omitted, there is always a most general way to recover the omitted type information. If there is no way to recover the type information, then the expression is ill-typed. Otherwise, there is a best way to "fill in the blanks", which will (almost) always be found by the compiler.

An interesting example, is the identity function:

$$fn\ x\ =>\ x$$

Since this function merely returns x as the result without performing any computation, there is no constraint on the type of x. Since the function is the same no matter which type is chosen, it is said to be *polymorphic*. Therefore, we say that the type of the identity function is $type \to type$.

There is clearly a pattern here, which can be understood by the notion of a *type scheme*: a type expression involving one or more type variables standing for an unknown. An *instance* of a type scheme is obtained by replacing each of the type variables occurring within the expression with a type scheme. For example, the type scheme $'a \to\ 'a$ has instances $int \to int$, $string \to string$, $(int * int) \to (int * int)$, and $('b \to\ 'b) \to ('b \to\ 'b)$, among infinitely many others. In contrast, it does not have the type $int \to string$ as an instance, since we are constrained to replace all occurrences of a type variable by the same type scheme. However, the type scheme $'a \to\ 'b$ has both $int \to int$ and $int \to string$ as instances.

Type schemes are used to denote the polymorphic behavior of functions. For example, the identity function could be written as follows.

$$\begin{aligned} val\ f1\ :\ 'a \to\ 'a\ &= \\ fn\ x :'\ a\ &=>\ x \end{aligned}$$

Note that we could have been far less explicit and wrote the following.

$$val\ f2\ =\ fn\ x\ =>\ x$$

While in practice it is easier to have ML infer the principle type, it is often good form to explicitly indicate the intended type, and this is very useful in debugging.

Often we need more than one type variable to express the polymorphic behavior of a function. For example,

$$\begin{aligned} val\ f3\ :\ 'a *' b \rightarrow\ 'b\ =& \\ & fn\ (x :\ 'a, y :\ 'b) => y \end{aligned}$$

is a function which returns the second element of any 2-tuple.

It is remarkable that every expression, with the exception of a few pesky examples), has a principle type scheme. That is, there is almost always a best or most general way to infer types for expressions. In this sense, every expression in ML seeks maximal depth. This is achieved through a process called *constraint satisfaction*. First, the expression determines a set of equations governing the relationships among the types of its subexpressions. Second, the constraints are solved using a process similar to Gaussian elimination, called *unification*. The equations can be classified by their solution sets (Linear Algebra anyone?) as follows:

i. Over-constrained: there is no solution. This corresponds to a type error.

ii. Under-constrained: there are many solutions. There are two sub-cases: ambiguous (overloading) and polymorphic (best solution).

iii. Uniquely determined: there is precisely one solution.

Overloading. Type information cannot always be omitted. The main source of ambiguity comes from the *overloading* of arithmetic operators. For example, in the function expression

$$fn\ x => x+x$$

it is not clear if the operator is with respect to integer addition or floating point addition. At this point it is clear that we are in the second situation of unification (see list above). The compiler has two options:

i. Declare the expression ambiguous, and force the programmer to provide enough explicit information to resolve the ambiguity.

ii. Arbitrarily choose a "default" interpretation, say the integer arithmetic.

Which choice does the SML/NJ compiler make?

Assignment Please do each of the following problems and put your code/comments in a file named *yourname_lab6.sml*. In addition, include the following commented line at the top of the file

(Name, Lab #, Date *)*

i. Include code for all expressions from the above reading, and describe the context of the example.

ii. Provide an example of an Over-constrained and Uniquely determined expression.

iii. Describe the difference, particularly as it pertains to how the compiler interprets the function double, in the following three expressions

a.

```
let
    val double = fn x => x + x
in
    (double 3, double 4)
end
```

b.

```
let
    val double = fn x => x + x
in
    (double 3.0, double 4.0)
end
```

c.

```
let
    val double = fn x => x + x
in
    (double 3.0, double 4)
end
```

iv. Explain why the following function declaration is ambiguous:

fun #name {name=n:string, ...} = n

However, we noted in Lab 3 that just such a predefined function, the sharp function, was defined by ML. Provide a conjecture on how the compiler removes the ambiguity in this function.
Hint: This is similar to how the ambiguity was removed from the function double in iii.