

# Dao Typing System

Limin Fu (phoolimin@gmail.com)

September 19, 2009

## Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Implicit Typing</b>	<b>2</b>
<b>3</b>	<b>Explicit Typing</b>	<b>4</b>

## 1 Introduction

The Dao typing system is a kind of blending between implicit and explicit, and static and dynamic typing, in that, the type of a variable can be defined explicitly or inferred implicitly, and it can be static for compiling time checking or dynamic for running time checking at the user's option. Language offering this feature is also called optionally typed. This kind of typing offers both the flexibility of dynamic-typed languages, and the safety of static-typed languages.

The Dao typing system can infer correctly the type of a variable in most cases (in other cases, the type is inferred as **any** which can hold any type of data), and check the type consistence of the operations before running the codes. Therefore, in Dao, one does not have to declare typed variables explicitly, nor need to worry about if it is safe not to do so. Moreover, it can exploit the typing information to generate some more specific and efficient virtual machine instructions for certain operations on numeric types. Sometimes, it may even create a new routine with some specialized instructions for a function call according to the parameter types of the call (resembling specialization of template functions in C++, or generic function in other languages)!

## 2 Implicit Typing

### 2.1 Basic Types

The simplest implicit typing happens when a variable is initialized with a constant value,

```
a = "a string"
b = 123
c = 456$
d = a + a
e = b + b
f = c + c

stdlib.debug();
```

Standard method `stdlib.debug()` will prompt to the user a debugging console where one can examine the compiled virtual machine codes (using **list** or **l** command), the data types and values of virtual registers (**about** or **a**) and the calling trace (**trace** or **t**) etc. One may observe from the debugging console that, *a* is inferred as *string*, *b* as *int* and *c* as *complex*; and respectively, the 3 addition operations are compiled as *ADD*, *ADDF* and *ADDC*.

### 2.2 List/Map/Tuple/Numeric Array

The type of a list/map/array is inferred by the types of its elements at its creation point. In subindexing, the type of the result is inferred according to the list/map/tuple/array type and the subindex type.

#### 2.2.1 List

```
a = { 1, 2, 3 } # list<int>
b = a[1]      # int, single element
e = a[1:1]    # list<int>, slicing, sublist

c = { 1, "A" } # list<any>
d = c[1]      # any
```

#### 2.2.2 Map

```
a = { "B"=>1, "A"=>2 } # map<string, int>
b = a["A"]           # int
```

```

b = a["A":"B"]          # map<string, int>, slicing, sub-map

c = { "B"=>1, "A"=>{} } # map<string,any>
d = c["A"]              # any

e = { { "B"=>1, "A"=>2 }, { "BB"=>1, "AA"=>2 } } # list<map<string,int>>

```

### 2.2.3 Tuple

```

a = ( "abc", 123, 10.55 ) # tuple<string,int,float>
a = ( name=>"abc", index=>123, value=>10.55 ) # tuple<name:string,index:int,value:float>

```

### 2.2.4 Numeric Array

```

a = [ 1, 2, 3; 4, 5, 6 ] # array<int>
b = a[0,2]              # int
c = a[1]                # int, treat "a" as a vector
d = a[1,]               # array<int>, slicing, second row
d = a[:,1:]             # array<int>, slicing, sub-matrix

```

Empty list/map/array are inferred as the corresponding types with an attribute indicating they are empty. Types with such attribute can be adapted to more specific types when necessary. For example,

```

a = { { 1, 2, 3 } }
a.append( {} );

stdio.println( a[1], stdlib.about( a[1] ) );

```

Where  $a$  is inferred as  $list<list<int>>$ , so the empty list appended to  $a$  will assume type  $list<int>$ .

## 2.3 Routine

```

routine foo( a ) # routine<a:?=??>
{
    b = a + a;
    stdlib.debug();
    return a;
}

```

```
stdio.println( stdlib.about(foo) );

foo( 1 ) # routine<a:int=>int>
foo( "ABC" ) # routine<a:string=>string>
```

Three copies of routines will be generated for *foo()*, with inferred types: *routine<a:?=>?>*, the original, *?* stands for undefined type; *routine<a:int=>int>*, specialized for call *foo( 1 )*; *routine<a:string=>string>*, specialized for call *foo( "ABC" )*. In routine types, each parameter typing has the form of: **type** (parameter of type **type** without name), **name:type** (parameter of type **type** with name **name**) or **name=type** (parameter of type **type** with a default value and named **name**). The returned type is indicated by **=>type**.

The addition instruction in the second copy is also specialized to take the advantage of inferred typing information. Parameter names are also incorporated into routine types for parameter checking in function calls with named parameters.

### 3 Explicit Typing

The type of a variable can be declared explicitly in the following form,

```
variable : type
variable : type = value
```

The same form can also be used in parameter list.

**type** can be one of the following keywords for built-in types: **int**, **float**, **double**, **long**, **string**, **complex**, **list**, **map**, **tuple**, **pair**, **array**, **buffer**, **routine**, and composition of them. **type** can also be a Dao class name or the name of a user-defined C type. Special keywords/symbols for typing include: **any** for any types, **?** for undefined types and **@X** for type holder that can be initialized to certain type in parameter list.

```
a : int
b : string = "ABC"
```

#### 3.1 Routine/Function Types

Function types are composed of **routine** with other types, they are usually inferred for functions from their declarations. But they can also be declared in the same way as other types, for example,

```
myfunc : routine<p1:string,p2=int=>int>
```

this will declare *myfunc* as a function that takes a string as the first parameter, a integer as the second with default value, and return a integer as result. See above for the typing of the parameters. In addition, `...` may also appear in the parameter list of a function type, it means variable number of parameters.

Example,

```
routine foo( a, b : string, c = 1 )
{
    return c;
}
stdio.println( stdlib.about( foo ) );
```

It must be noted that the type matching of function types is different from other types. When a function type is assigned to another function type, or passed to a function call with another function type as parameter, a "narrower" function type can not be matched to a "wider" function type.

Consider the following situation,

```
routine foo( a : int ){ ... }

routine bar( callback : routine<any=>any> ){ callback( "ABC" ) }

bar( foo );

myfunc : routine<any=>any> = foo;
myfunc( "ABC" );
```

If the "narrower" function type of *foo* may match to the "wider" function type *routine<any=>any>*, there will be a problem when *bar* is called with *foo* as the callback. Because *bar* expects that the callback can take any type as parameters, so it may invoke the callback with a string, while *foo* can only accept a integer! The problem for *myfunc* is similar. So for type matching between function types, it is always that a "wider" function type can be matched to a "narrower" function type!

How about if one wants a function to accept a parameter of certain type and another parameter of the same or a related type, or it returns a such type? It can be done by using a type holder in the form of `@X`. For example, *foo* in the following example will take a list as its first parameter, and need the second parameter and returned value to be the type of the items of the list, one can do as,

```
routine foo( mylist : list<@A>, it : @A ) => @A
{
  #do something
}
```

---

Then the typing system will ensure that only consistent types can be passed to *foo* .