Dao Programming Language for Scripting and Computing

Limin Fu

September 18, 2013

StrangeLoop 2013: Emerging Languages Camp

Outline

Overview: Development Motivation

Initial motivation: Perl!

My frustration with Perl made me very curious about language design and implementation.

Another motivation: Bioinformatics.

I really wanted a better porgramming language for bioinformatics.

Now the goal is to create a general purpose language

that offers advanced features supported by a small runtime, with emphasis on:

- rich but non-redundant data types and features;
- consistent and reasonably designed syntax;
- simple programming interfaces for extending and embedding;
- good efficiency for numeric computation;
- good support for multicore machines;

Overview: Syntax Style and Performance Expectation

What kind of syntax can you expect from Dao?

```
# With implicit types:
routine Sum( nums )
{
    sum = 0
    for (x in nums) sum += x
    return sum
}
ints = {1 : 2 : 1000}
fut  = Sum( ints ) !!
sum  = fut.value()
```

```
# With explicit types:
routine Sum( nums : list<int> ) => int
{
    sum : int = 0
    for (x in nums) sum += x
    return sum
}
ints : list<int> = {1 : 2 : 1000}
fut  : future<int> = Sum( ints ) !!
sum  : int = fut.value()
```

Overview: Syntax Style and Performance Expectation

What kind of syntax can you expect from Dao?

```
# With implicit types:
routine Sum( nums )
{
    sum = 0
    for (x in nums) sum += x
    return sum
}
ints = {1 : 2 : 1000}
fut  = Sum( ints ) !!
sum  = fut.value()
```

```
# With explicit types:
routine Sum( nums : list<int> ) => int
{
    sum : int = 0
    for (x in nums) sum += x
    return sum
}
ints : list<int> = {1 : 2 : 1000}
fut  : future<int> = Sum( ints ) !!
sum  : int = fut.value()
```

What kind of performance can you expect from Dao? *(Time in seconds)*

Program	Argument	Dao	Dao+JIT	Speedup	Lua	Python	C (-O2)
fannkuch	11	59.3	16.0	3.7X	135.1	279.0	2.9
mandelbrot	4000	24.1	4.3	5.7X	55.8	132.1	2.3
nbody	10000000	35.7	11.9	3.0X	93.2	261.4	1.7
spectral-norm	5000	20.5	2.0	10.4X	69.4	287.1	1.9
binary-trees	16	30.4	30.3	1.0X	20.7	19.7	4.5
meteor	2098	5.5	5.5	1.0X	2.1	9.6	0.1

Overview: Feature Lists

Key Features

- **Optional typing** with **type inference** and static type checking;
- BNF-like **syntax macro** for defining customized syntax;
- Native support for **concurrent programming**;
- LLVM-based **Just-In-Time (JIT) compiling**;
- **Simple C interfaces** for easy embedding and extending;

Other Main Features

Modules and Tools

Overview: Feature Lists

Key Features

Optional typing, syntax macro, concurrent programming, JIT compiling etc.

Other Main Features

- Has **enum symbols**, **tuples**, **numeric arrays** and **hash maps** etc.;
- Object-Oriented Programming (**OOP**) with **classes** and **interfaces**;
- Support **mixin** class, **class decorator** and **aspect class** (for AOP);
- Support **coroutines**, **decorators**, **anonymous functions** and **closures**.
- **Code section methods** as an alternative to functional methods;
- Built-in support for **string pattern matching**;
- Template-like C data type;
- Designed and implemented as a **register-based virtual machine**;
- Bytecode file format, archive file format and single file deployment.

Modules and Tools

Overview: Feature Lists

Key Features

Optional typing, syntax macro, concurrent programming, JIT compiling etc.

Other Major Features

Rich data types, OOP, mixins, aspects, coroutines, decorators, anonymous functions, closures, code section methods and string pattern matching etc.

Major Modules and Tools

- Standard online **help** system;
- Standard module for data **serialization**;
- Clang-based module to support **mixing C/C++ code** with Dao code;
- ClangDao: Clang-based tool for **automatic wrapping** of C/C++ libraries;
- DaoStudio: Integrate Developement Environment;

Optional Typing in Dao

Typical places to use optional types:

- **Variable declaration:**

```
tup : tuple<int,string> = ( 123, 'abc' )
tup                     = ( 123, 'abc' )
```

- **Function declaration:**

```
routine Test( a : float = 0.0 ) => int { return a > 1.0 }
routine Test( a         = 0.0 )        { return a > 1.0 }
```

- **Class definition:**

```
class SimpleClass
{
    var  index : int    = 0
    var  name  : string = 'Joe'
}
class SimpleClass
{
    var  index  = 0
    var  name   = 'Joe'
}
```

Optional Typing in Dao

Type System

Optional typing is enabled by a very simple type system, which does (mostly) **instruction-wise** type inference and static type checking.

Additional features enabled by this simple type system

- **Instruction specialization:**
- **Function specialization (at both compiling and running time):**
- **Boilerplate code saving:**

Optional Typing in Dao

Type System

Optional typing is enabled by a very simple type system, which does (mostly) **instruction-wise** type inference and static type checking.

Additional features enabled by this simple type system

- **Instruction specialization:**

```
a = 123
b = a + 456
```

  The addition in the above code will be compiled into **ADD**, then specialized to **ADD_III** for integer type operands after type inference.

- **Function specialization (at both compiling and running time):**
- **Boilerplate code saving:**

Optional Typing in Dao

Type System

Optional typing is enabled by a very simple type system, which does (mostly) **instruction-wise** type inference and static type checking.

Additional features enabled by this simple type system

- **Instruction specialization:**
- **Function specialization (at both compiling and running time):**

```
routine Test( a ){ return a + a }

Test( 123 )
Test( 'abc' )
```

The `Test(a)` function will be specialized at compiling time into two functions `Test(a:int)` and `Test(a:string)` according to the calling parameters.

- **Boilerplate code saving:**

Optional Typing in Dao

Type System

Optional typing is enabled by a very simple type system, which does (mostly) **instruction-wise** type inference and static type checking.

Additional features enabled by this simple type system

- **Instruction specialization:**
- **Function specialization (at both compiling and running time):**
- **Boilerplate code saving:**
  To wrap C function `int test(float a)` for Lua or Python, one has to write *boilerplate code to check the parameter types*. But **not in Dao**,

```
void dao_test( DaoProcess *proc, DaoValue *par[], int n )
{
    float a = par[0]->xFloat.value;
    DaoProcess_PutInteger( proc, test( a ) );
}
```

  Parameter type checking is not necessary if the wrapping function is registered with a **proper signature**,

```
DaoNamespace_WrapFunction( ns, dao_test, "test(a:float)=>int" );
```

Dao Syntax Macro

Basic idea:

Syntax of programming languages can often be specified by the (extended) Backus Normal Form (BNF).

- A **BNF expression** can be view as a **pattern** for both **matching** and **generating** token sequences;
- Combining two BNF (like) expressions, one for **matching** and the other for **generating** token sequences, you get a **BNF-like syntax macro**;

Dao Syntax Macro

Basic idea:

Syntax of programming languages can often be specified by the (extended) Backus Normal Form (BNF).

- A **BNF expression** can be view as a **pattern** for both **matching** and **generating** token sequences;
- Combining two BNF (like) expressions, one for **matching** and the other for **generating** token sequences, you get a **BNF-like syntax macro**;

Dao Syntax Macro

In Dao, syntax macro can be expressed in the following way,

```
syntax [ optional_language_id ] {
    source_syntax_pattern
} as {
    target_syntax_pattern
}
```

Dao Syntax Macro

Example

The following tokens are controlling markers,

- **( )** : pattern grouping;
- **! ? * +** : group repeating;
- **[ ]** : optional group, equivalent to **( ) ?**;

and tokens started with **$** are special variables,

- **$ID** : a valid identifier;
- **$EXP** : an expression or subexpression;
- **$BL** : a block of code may contain any type of syntax structures;

```
syntax { # while do end
    |'while' $EXP1 'do' [ $BL1 ] 'end'
} as {   # while(){}
    |'while' '(' $EXP1 ')' '{' [ $BL1 ] '}'
}

a = 1
while a < 10 do
    |io.writeln( a )
    |a += 1
end
```

Dao Code Section/Block Method

Code Section/Block Methods

Code section method is a special type of method that can take a block of code as an implicit parameter when called. The code block is attached to the call by `::{ [param_name] code_block }`.

- Builtin code section methods:

```
ints = { 123, 987, 376, 345 }
ints.iterate( $backward )::{ [item,index]
    io.writeln( item, index )
}
```

- User defined code section methods:

```
routine Test( maxvalues : list<float> )[ float => float ]
{
    for( max in maxvalues ){
        r = yield( max )  # execute the code section;
        io.writef( 'Random value in [0,%g]: %g\ n', max, r )
    }
}
Test( { 100.0, 200, 300 } )::{ [X] rand( X ) }
```

Dao Decorator, Mixin and Aspect

Decorator

Decorators are functions that can modify (decorate) other functions:

```
routine @Decorator( func : routine ) {
    io.writeln( 'Calling function:', std.about(func) );
    return func( __args__, ... ); # ... for parameter expanding;
}
@Decorator
routine Function() { io.writeln( 'Function()' ); }
```

Dao Decorator, Mixin and Aspect

Decorator

Decorators are functions that can modify (decorate) other functions:

```
routine @Decorator( func : routine ) {
    io.writeln( 'Calling function:', std.about(func) );
    return func( __args__, ... ); # ... for parameter expanding;
}
@Decorator
routine Function() { io.writeln( 'Function()' ); }
```

Mixin

Mixins are classes with members injected from component classes without inheritance:

```
class Component {
    var value = 456
    routine Meth2(){ io.writeln( self, value ) }
}
class Mixin ( Component ) {
    var index = 123
    routine Meth(){ io.writeln( self, index, value ) }
    routine Meth2( a : string ){ io.writeln( self, index, value, a ) }
}
```

Dao Decorator, Mixin and Aspect

Class Decorator

Class decorators are classes whose decorator methods will be automatically applied to mixin classes:

```
class @DecoratorClass {
    routine @Prefix( meth :routine<self:@Decorator> ) for Prefix~ {
        io.writeln( 'Decorator::Prefix()' )
        meth( __args__, ... );
    }
}
class MyMixin ( @DecoratorClass ) {
    routine PrefixTest() {
        io.writeln( 'MyMixin::PrefixTest()' )
    }
}
```

Dao Decorator, Mixin and Aspect

Class Decorator

Class decorators are classes whose decorator methods will be automatically applied to mixin classes:

```
class @DecoratorClass {
    routine @Prefix( meth :routine<self:@Decorator> ) for Prefix~ {
        io.writeln( 'Decorator::Prefix()' )
        meth( __args__, ... );
    }
}
class MyMixin ( @DecoratorClass ) {
    routine PrefixTest() {
        io.writeln( 'MyMixin::PrefixTest()' )
    }
}
```

Aspect

Aspects are decorator classes to be applied to other classes automatically:

```
class @AspectForMyClasses for My~ {
    routine @Method( meth : routine ) for Method~ {
        io.writeln( 'In @AspectForMyClasses::@Method():', std.about(meth) )
        return meth( __args__, ... )
    }
}
```

Concurrent Programming in Dao

Dao has multiple features to support concurrent programming

- Asynchronous Function Call;
- Asynchronous Object;
- Tasklet communication channel;
- Built-in multithreading module **mt**;
- Concurrent garbage collector;

Concurrent Programming in Dao: Asynchronous Function Call

Asynchronous Function Call

- Is a call followed by **!!**;
- Executes in a **separated tasklet** (a very lightweight thread);
- Returns a **future value** (for scheduling and/or retrieving results);

Example

```
routine SumOfLogs( n = 10 )
{
    sum = 0.0
    for( i = 1 : n ) sum += log( i )
    return sum
}

fut = SumOfLogs( 2000000 ) !!    # Asynchronous mode;

while( fut.wait( 0.01 ) == 0 ) io.writeln( 'still computing' )
io.writeln( 'sum_of_logs =', fut.value() )
```

Concurrent Programming in Dao: Asynchronous Object

Asynchronous Object

- An asynchronous object is a class instance created in asynchronous call mode;
- All its methods will be invoked asynchronously (execute in tasklets and return future values);
- Such tasklets are scheduled such that at most one thread task is active for the same instance at any time.

Example

```
class Clustering
{
    routine Run(){ DoKmeansClustering() }
}
cls = Clustering() !!     # Asynchronous mode;
job = cls.Run()
while( 1 ){
    DoSomethingElse();
    if( job.wait( 0.1 ) ) break; # wait for 0.1 second
}
```

Concurrent Programming in Dao: Channel

Tasklet Communication Channel

- Channel allows passing data and synchronizing between tasklets;
- The channel type is implemented as a customized C data type that supports template-like type arguments:

```
chan = mt::channel<int>( 5 )  # integer channel with capacity 5;
```

- It provides two key methods among others:

```
send( self :channel<@V>, data :@V, timeout :float = -1 ) => int

receive( self :channel<@V>, timeout :float = -1 )
  => tuple<data :@V|none, status :enum<received,timeout,finished>>
```

Concurrent Programming in Dao: Channel

Example

```
class Producer
{
    routine Run( chan : mt::channel<int> ){
        index = 0;
        while( ++index <= 100 ) chan.send( index )
        chan.cap(0)  # set channel buffer size to zero to close the channel;
    }
}
class Consumer
{
    routine Run( chan : mt::channel<int> ){
        while(1){
            data = chan.receive()
            io.writeln( "received", data );
            if( data.status == $finished ) break
        }
    }
}
chan = mt::channel<int>(2)
producer = Producer() !!
consumer = Consumer() !!
producer.Run( chan )
consumer.Run( chan )
```

Concurrent Programming in Dao: Multithreading Module mt

Concurrent Programming with mt Module

mt is a built-in module to provide additional multi-threading functionalities. It can be used to create tasklets, but more importantly, it offers **parallelized code section methods** to make certain parallelization much simpler.

Tasklet and future value

Tasklet can be created with **mt.start()::{}**, and handled with a **future value** type.

```
# Start a thread task and return a future value:
fut = mt.start( $now )::{
    sum2 = 0
    for( i = 1 : 1000 ) sum2 += i * i
    return sum2
}
while( fut.wait( 0.01 ) == 0 ) io.writeln( 'still computing' )
io.write( 'sum_of_squares = ', fut.value() )
```

Concurrent Programming in Dao: Multithreading Module mt

Parallelized code section methods

- **mt.iterate()**: iterate on array, list, map, or just a number of iteration;
- **mt.map()**: map items of array, list or map to produce new array or list;
- **mt.apply()**: apply new values to the items of array, list or map;
- **mt.find()**: find the first item that satisfy a condition;

Example,

```
ls = {1,2,3,4,5,6}

# Concurrent iteration:
mt.iterate( times => 10, threads => 4 )::{ io.writeln( X ) }
mt.iterate( ls, threads => 4 )::{ io.writeln( X ) }

# Parallelized mapping and value application:
ls2 = mt.map( ls, 4 )::{ X*X }  # ls2 = {1,4,9,16,25,36}
mt.apply( ls, 4 )::{ X*X }      # ls  = {1,4,9,16,25,36}

# Parallel searching:
num = mt.find( ls, 4 )::{ X > 20 }
```

LLVM-based Just-In-Time (JIT) compiler

- Implemented as a loadable module (DaoJIT);
- Backend based on the **LLVM**;
- Emphasis on **numeric computation**;
- Compiles a subset of Dao virtual machine instructions;

LLVM-based Just-In-Time (JIT) compiler

- Implemented as a loadable module (DaoJIT);
- Backend based on the **LLVM**;
- Emphasis on **numeric computation**;
- Compiles a subset of Dao virtual machine instructions;

JIT Performance Test (time in seconds)

Program	Argument	Dao	Dao+JIT	Speedup	Lua	Python	C (-O2)
fannkuch	11	59.3	16.0	3.7X	135.1	279.0	2.9
mandelbrot	4000	24.1	4.3	5.7X	55.8	132.1	2.3
nbody	10000000	35.7	11.9	3.0X	93.2	261.4	1.7
spectral-norm	5000	20.5	2.0	10.4X	69.4	287.1	1.9
binary-trees	16	30.4	30.3	1.0X	20.7	19.7	4.5
meteor	2098	5.5	5.5	1.0X	2.1	9.6	0.1

Note 1: benchmark programs are taken from *Computer Language Benchmarks Game* `http://shootout.alioth.debian.org`; **Note 2:** the last two are not JIT compiled, because they don't contain enough JIT compilable code (for the current JIT compiler).

ClangDao: bringing C/C++ libraries to your finger tips

- Based on **Clang** (C Language Family Frontend for LLVM);
- Generate bindings directly from C/C++ **header files**;
- Support C/C++ functions, C structs, C callbacks, C++ classes and inheritance, C++ virtual functions, C++ templates (to some extent) etc.;
- Support user-defined wrapping hints expressed as C macros;

ClangDao: bringing C/C++ libraries to your finger tips

- Based on **Clang** (C Language Family Frontend for LLVM);
- Generate bindings directly from C/C++ **header files**;
- Support C/C++ functions, C structs, C callbacks, C++ classes and inheritance, C++ virtual functions, C++ templates (to some extent) etc.;
- Support user-defined wrapping hints expressed as C macros;

Example input file for ClangDao

- File *mymodule.c*:

```
#define module_name MyModule
#undef module_name
// Hint to mark a pointer parameter as an array of size 3:
#define dao_mytest( p_dao_hint_array_3 ) mytest(int*)
// Constants, functions and classes etc. from the included
// header files will be wrapped:
#include "myheader.h"
```

- Then the bindings can be generated with:

```
$ clangdao -IPathToHeaderFile mymodule.c
```

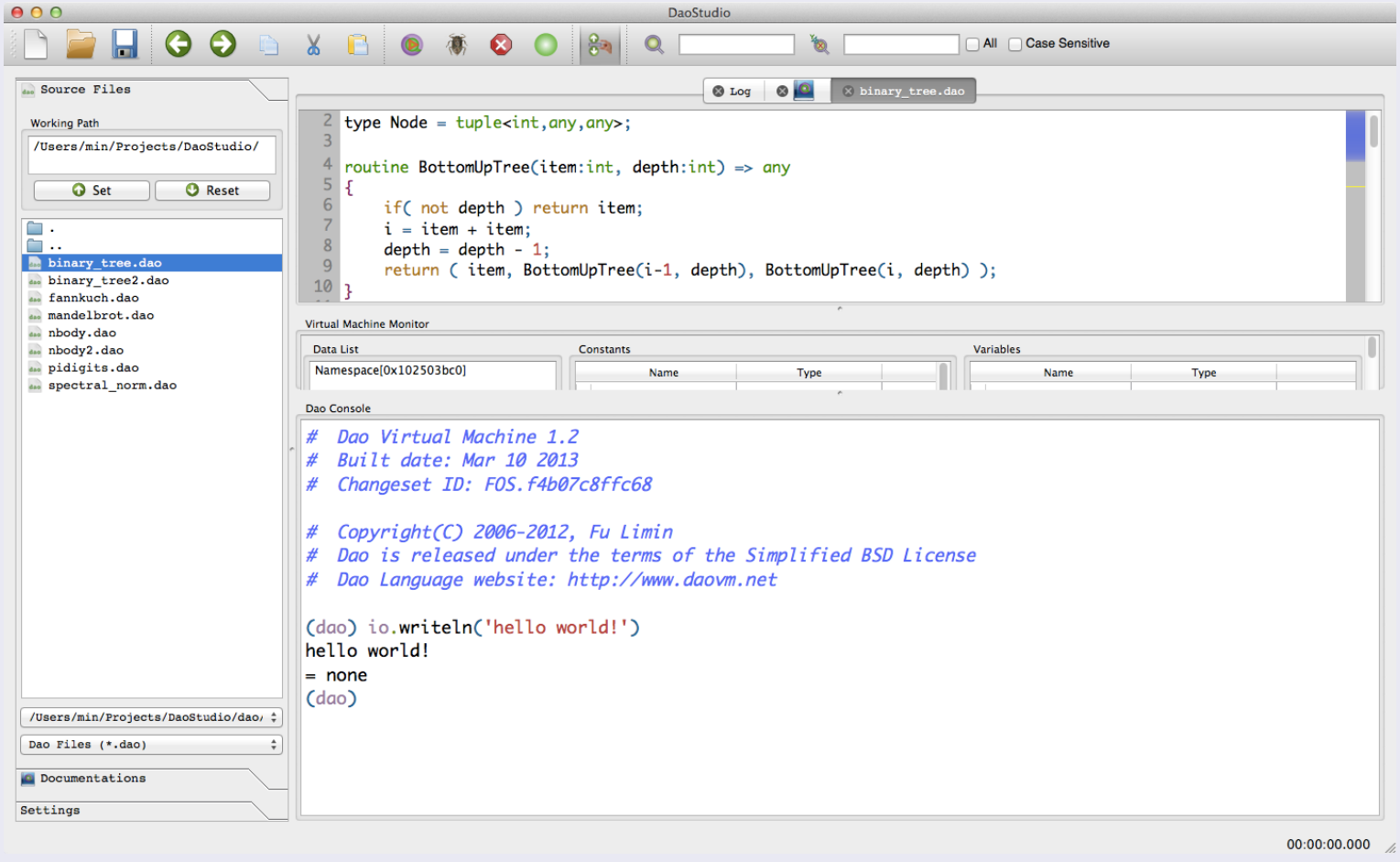
ClangDao: bringing C/C++ libraries to your finger tips

List of bindings generated by ClangDao

Scientific:	Dao**GSL**	GNU Science Library (GSL)
	Dao**BamTools**	BamTools
	Dao**GenomeTools**	GenomeTools
	Dao**SVM**	LibSVM (Support Vector Machine)
Visualization:	Dao**VTK**	Visualization Toolkit
	Dao**MathGL**	MathGL
2D Graphics:	Dao**GraphicsMagick**	GraphicsMagick
3D Graphics:	Dao**OpenGL**	OpenGL
	Dao**Horde3D**	Horde3D Engine
	Dao**Irrlicht**	Irrlicht 3D Engine
Multimedia:	Dao**SDL**	Simple DirectMedia Layer (SDL)
	Dao**SFML**	Simple and Fast Multimedia Library
GUI:	Dao**FLTK**	Fast Light Toolkit (FLTK)
Miscellaneous:	Dao**XML**	libxml2
	Dao**Bullet**	Bullet Physics Engine
	Dao**GameKit**	GameKit Game Engine
	Dao**GamePlay**	GamePlay Game Engine

DaoStudio: Integrate Development Environment for Dao

A cross platform IDE for Dao developed using Qt4



Future Development

Main Development

- Better documentations for the language, modules, and tools;
- Possible improvements to the implementation;
- Development of comprehensive unit tests;
- Further improvements to the JIT compiler;
- Further improvements to the ClangDao tool;
- Further improvements to the DaoStudio IDE;

Acknowledgements

Thanks to *Aleksey Danilov*, *Lucas Beyer*, *Belousov Oleg*, *Zhiguo Zhao* and others for module contributions, testing or bug reports etc.

Thank you for your time!

Homepage: http://daovm.net