Haplo-Kernel tutorial

Nicolas Laurent

December 12, 2002

Abstract

Haplo come with a specific language. This dialect is very simple and can be easily interfaced with dynamic library. If the library in written in C, the binding is quasi-immediate. Things are little more complex with C++ or other language framework. This paper introduce haplo-language and the way to bind a C framework with it.

1 The haplo-language

1.1 Basic concepts

In haplo, there're two basic components: objects and functions. The functions are always binded with a C-function in one dynamic library. For example you will see that `cos` is (nearly) directly mapped to `cos` from "`libm.so`".

The other component is object. Each object have a type. The number of different types is dynamic and can change with importation of dynamic libraries. As function, type is binded with C-type more or less elaborated. For example the type `<string>` is directly mapped to `char *`. Any object can be referenced by a name called identifier. A identifier could be the name of a function, then it shadows the function. The only forbidden names to give to an object are the keyword of the language.

1.2 The interface

Haplo is designed to run in interactive session. But it is possible to write your program in a text editor and launch them in batch mode. The one thing you should know is that each instruction must be separeted by a "`;`" from the others. A instruction could be a assignation : "`a=1.57`" or a call to a function : "`cos(1.57)`". So "`a=1.57; cos(a)`" is perfectly legal.

Everytimes Haplo is readly to get more instruction, it will display a prompt: "`>>`". You could type one instruction in multiple lines, remind simply that you should terminate it by a "`;`". Each time Haplo can interpret a instruction it display the result.

Comments begin with "`#`". When haplo encounter a "`#`" it will ignore the rest of the line.

1.3 The keywords

There're 12 keywords in haplo-language. This keywords cannot be used as identifiers nor be overloaded by function declaraion. For the majority, their meaning are the same as the C keyword. I won't explain them in this paper.

break	**continue**	**else**
for	**free**	**function**
if	**info**	**load**
quit	**return**	**while**

1.4 The basic types

The standard **haplo-kernel** comes with 6 builtin types of objects:

<boolean> <code> <float>
<library> <string> <vector>

The object of type <float> can hold value in the interval that C `double` can do. You can construct object of this type by typing any number. There're no type of integer value in Haplo. 3.14, 2 and -15e-230 are valid values.

The objects of type <string> can be constructed by unsing "`"`" arround any text. Just notice you cannot enter a string on multiple line.

The objects of type <boolean> can hold one of the two values: true or false. You cannot construct this kind of object directly, you should use something like "`a = (1==1);`" for example (the parenthesis are optional be simplify the read). But there certainly exit objects named "`true`" and "`false`" defined in the standard environnement.

The objects of type <library> are just handlers of an imported dynamic library which are not stricly haplo modules. We will use it later.

The objects of type <vector> are aggregations of whatever objects. You can contruct them using "`[`" and "`]`". Every members of a vector are separated by a "`,`". You could extract a component of a vector with "`->`". Rememeber that the first element of a vector is numbered as 1 (and not 0 as C-style). This is a simple example of what you can do :

```
>> vec=[ 1, 2.0, "a string" ];
   Vector(3)
>> vec->2
   2.0
```

Finally the objects of type <code> are runtime-defined functions. They are not functions as described in the begining of the document but their behaviour is very close. You can construct objects of type <code> by using the keyword "**function**". The return value is the result of the last interpreted instruction. There's a simple example :

```
>> f=function(a) { a+2; };
   Code (3 Ops, Internal references)
>> f(2);
   4
```

You could see that this <code> is displayed by Haplo as "`Code (3 Ops, Internal references)`". The informations inside parenthesis are not very useful by now, they just be interresting for optimisation.

Keep in mind that <code> are simple objects like <float> or <string> are. For example you can pass object of type <code> as parameter of other one :

```
>> f1=function(f) { f(2); };
   Code (3 Ops, Internal references)
>> f2=function(a) { a+2; };
   Code (3 Ops, Internal references)
>> f1(f2);
   4
```

<code> objects are very powerful. They do not need to know what will be the type of their parameters and the type of their return value can be anything. All of that will work because many function are overloaded.

1.5 Function overloading

Function can be overloaded. This can mean two things:

- a function can have more than one prototype. This is true for the function print. You can get all the prototypes of a function by the keyword **info**:

```
>> info print;
        void print(const string);
        void print(const float);
        void print(const code);
```

  At runtime, the prototype that matches is search. If one is found, the function is called. If not, a error occured and a message is issued. Prototypes differ each others by the number and the type of their argument but not by their return value. Two prototype for the same function can exist at the same time but the last declared shadows the first. We will see how to declare a function in section 2.

- a object can have the name of a function. In this case, when the function is called, the object replace it. This have some sense when the object have type <code>. Oject always takes hands over functions.

1.6 Some examples

There's some examples of what you can do. More examples can be found in ""`tests`"" directory of the sources tree.

Factorial

This example shows the keyword **for** in action.

```
>> fact=function(n)
>> {
>>      result=1;
>>      for(i=2; i<n+1; i=i+1)
>>              result = result * i;
>>      result;
>> };
   Code (21 Ops, Internal references)
>> fact(7);
   5040
```

Recursive factotrial

This example is more complex than the previous since it implements recursivity.

```
>> fact=function(n)
>> {
>>      if (n>1)
>>              fact(n-1)*n;
>>      else
>>              1;
>> };
   Code (12 Ops, Internal/External references)
>> fact(7);
   5040
```

2 Howto to bind a C Library with Haplo

2.1 Create a new function

Consider the following piece of frame work. "compute" is one of public functions.

"libtest.c"

```
#include <stdlib.h>

double *compute(const double *value1, const double *value2)
{
        double *result;

        result=malloc(sizeof(*result));

        *result=do_some_work(*value1, *value2);

        /* ... */

        return result;
}
```

"libtest.c"

There're two ways of making "compute" available from from Haplo.

Using haplo language

If the datatypes for parameters and return value of the function already exist in Haplo, it's very simple to do:

```
>> lib=load_library("libtest.so");
   library
>> bind(lib, "compute", "computation", "float:float:float");
>> computation(2.3, -5.2);
   75.3
```

Using haplo API

The other method is to use Haplo API. So you will need to compile and link this with your framework. Don't forget to link it with "`haplo -libs`".

"libtest-init.c"

```
#include <haplo.h>

extern double *compute(double *, double *);

int test_haplo_init(int haplo_major, int haplo_minor)
{
        haplo_type_t type_float;

        type_float=HAPLO_OBJECT("float");

        HAPLO_FUNC_REGISTER_2(compute, "computation", type_float,
                type_float, HAPLO_ARG_IN,
                type_float, HAPLO_ARG_IN);

        return(HAPLO_SUCCESS);
}
```

"libtest-init.c"

With this little piece of code, ""`libtest.so`"" is become a *haplo module*. We do not need to play with library handle, nor to bind functions. We just need to load this module:

```
>> use("test");
>> computation(2.3, -5.2);
   75.3
```

2.2 Create a new-datatype