Elementary Analysis Of TinyPython Virtual Machine v1.1

Hari John Kuriakose, Prasanth Madhavan, Sumith S., Sunil Kumar

September 10, 2010

Abstract

A basic analysis of source code for TinyPy v1.1, excluding the byte-code compilation phase. The focus is on the working of the TinyPy Virtual Machine, which actually executes the byte-code.

Overview of TinyPython

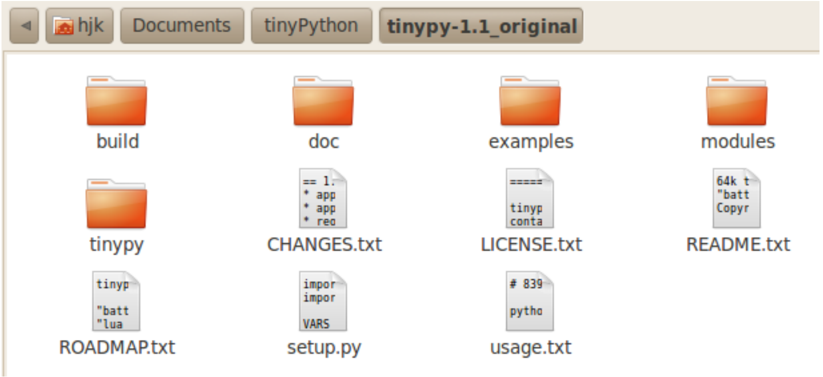
TinyPython is a minimalist implemenation of Python in 64K code. It is a parser and byte-code compiler written in TinyPython itself. It is also fully bootstrapped in the sense that initially, TinyPython converts a Python script (.py) into a special TinyPy byte-code format (.tpc), and this generated code is then passed into a subset of the TinyPython source code called the Virtual Machine, where the actual execution takes place. One can even extend the idea that if the VM is compiled into a low-level format adaptable to a particular microcontroller, then the VM will reside inside that chip, and any .tpc file can be downloaded into the chip as its input.

As outlined above, TinyPy comprises of two phases: the byte-code generation, and the execution of the byte-code in the TinyPy Virtual Machine. Out of these, the first phase will not be discussed here.

Building Up

The TinyPython source code used was downloaded from here.

Initially, the file listing will look like the following figure.



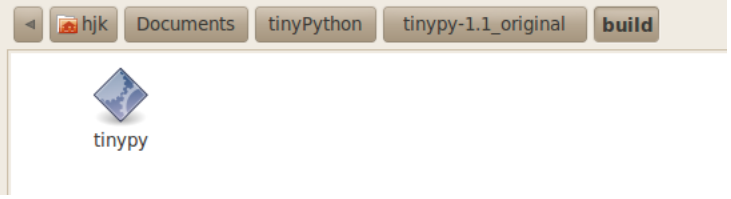
You can find that the 'build' folder will be empty. The 'doc', 'examples' and 'modules' folder may or may not contain any documents, Python scripts and batteries (or modules) respectively, depending on the downloaded package. The LICENSE.txt and README.txt are self-explanatay names. The CHANGES.txt contain a record of all the changes that the autho of TinyPy thought of making in the source code at some point. The ROADMAP.txt gives a brief description of the features of TinyPy, and an idea about the future developments to be implemented. The 'setup.py' contain the initial code to build the TinyPy from scratch.

At the terminal, type as follows:

```
python setup.py linux
```

It is implied that you need a Python interpreter available in your system. The 'linux' option is to specify that the code will be compiled so as to make it work in a linux environment.

After running this command, a new executable 'tinypy' will appear in the 'build' folder as shown.



To fully bootsrap and test TinyPy, give the command as,

```
python setup.py linux boot
```

Now, in the 'tinypy' folder shown in the above figure, two new executables, 'tinypy' and 'vm' will appear, which are the TinyPy parser, and Virtual Machine respectively.

It can be noticed that all the .pyc files for the corresponding .py files have been generated by the Python interpreter. In addition to that, some test scrips - named as tmp - will be invoked too. The most interesting thing will be the presence of new .tpc files for some of the initial .py files.

The general usage and some options available are listed below.

```
python setup.py command [option] [module]
```

- 64 k - build a a64k version of TinyPy source code
- blob - build a single tinypy.c and tinypy.h

The tinypy folder will now have the following contents:

Out of these, the 'py2bc.py' script is used to convert a user-generated Python script into its corresponding .tpc file. The format will be:

```
<tinypy_path> py2bc.py sample.py sample.tpc
```

Here, tinypy_path is the path (relative to current position) of either the tinypy executable in the 'build' folder, or the one in the 'tinypy' folder. 'sample.py' is the name of the user-script. 'sample.tpc' is the name given for the byte-code converted file. Or you can simply give it as:

```
python py2bc.py sample.py sample.tpc
```

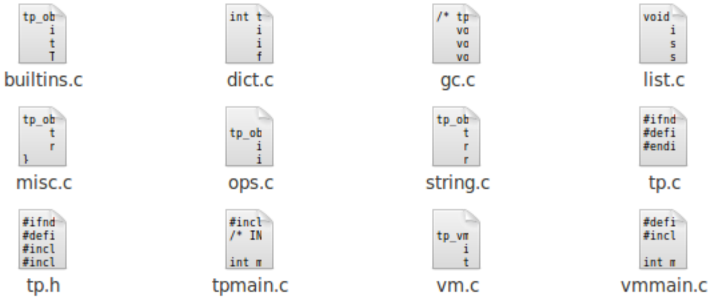
Finally, the generated byte-code (.tpc) is to be passed into the VM for compilation and execution. Assuming the current directory as 'tinypy' folder, it is done as:

```
vm sample.tpc
```

Or logically,

```
gcc vmmain.c -lm
./a.out sample.tpc
```

The 'vmmain.c' will be present in the 'tinypy' folder. It is the main function of the VM which runs and automatically links to all the other files necessary for the VM. It is necessary to link the math module too, hence the option '-lm'. And now the output is obtained and displayed. For a better picture, the files needed for VM are:



Writing and compiling the code only accounts to half of the process. The other half is debugging and understanding the flow of control within the source code. To do that, make use of the GNU debugging tool, 'gdb'.

```
gcc -g vmmain.c -lm
gdb ./a.out
```

Inside the 'gdb', you can set breakpoint for any function. Then run the process for the byte-code you need. Here, 'sample.tpc' is used as example.

```
(gdb) run sample.tpc OR r sample.tpc
```

Another essential tool will be 'ctags'. After its installed, go to the 'tinypy' folder and build the tag stack as follows:

```
ctags *.c *.h
```

You can see that a new file named 'tags' is now available. Now when you are inside a .c or .h file, you can use 'ctrl +]' and 'ctrl + T' to jump back and forth between cross references spanning different files.

Analyzing the Virtual machine

The major task that we had to perform was to analyze the working of the virtual machine. Analyzing and understanding its soure code was, obviously, the major challenge that we faced. The code itself was not well arranged and the lack of proper documentation made the challenge even bigger. Proper use of the GNU debugger, 'gdb', and the indexing tool, Ctags made the process of analysis of the VM a lot simpler.

Perhaps the best way to understand the working of the VM is to use the VM to execute the byte code of a simple python program and see and uderstand the actual flow of control. Here we use a sample program 'sample.py':

```
def add(a,b):
    c = a + b
    return(c)
print(add(1,9))
```

Note that the use of the parentheses for built-in functions too is a part of the syntax in TinyPy.

The .tpc (byte code) for sample.py is obtained as already-stated.

```
cc vmmain.c -lm
./a.out sample.tpc
```

gives you the output:

```
10
```

Understanding the flow of control for this program would give us an idea of the working of a segment of the virtual machine.

Each time the VM is invoked, a new instance of the VM is created .

```
tp_vm *tp =tp_init(argc,argv);
```

The key point to be understood in the analysis of VM is that each object in tinypy is stored as a union in the C API. Lets get into the details:

tp_obj is the tinypy's object representation.

```
typedef union tp_obj{
int type;
tp_number_ number;
struct{int type;int *data;}gci;
tp_string_ string;
tp_dict_ dict;
tp_list_ list;
tp_fnc_ fnc;
tp_data_ data;
}tp_obj;
```

The union tp_obj with its fields are shown. The field 'type' of the union indicates the type, of the object. A type value of 1 indicates that a number is being used, type = 2 indicates object is a string and type = 4 indicates a list.

In our sample program, our objective is to add two numbers 'a' and 'b' and print their output.

A function in the virtual machine 'tp_add' does the job of adding two arguments that is passed to the function as input. The arguments could be numbers, strings or lists. Lets analyze the function 'tp_add'.

```
tp_obj tp_add(TP,tp_obj a,tp_obj b)
```

The Function defintion of 'tp_add' contains three arguments:

TP -> the VM instance

tp_obj a -> the union variable representing the object 'a' in the sample program.

tp_obj b -> the union variable representing the object 'b' in the sample program.

Within the function tp_add, the type of the two objects are checked and the appropriate function is performed.

In our example ,two numbers are to be added.

As we mentioned above, 'a' and 'b' are stored as unions and the union contains fields for types such as number, strings, lists etc. We access the field tp_number_ in the union 'a' as a.number. 'number' is a variable of the structure tp_number_ which contains a variable 'val' that stores the exact value of the number . Therefore a.number.val would give you the actual object .

Analyzing the addition of strings would be interesting as well. 'a.string' would give you the union which represents the string object. The union tp_obj contains a field tp_string_ which is a structure and includes the pointer that points to the address where the string 'a' is stored.

The structure tp_string_:

```
typedef struct tp_string_{
int type;
struct tp_string_ info*;
char const *val;
int len;
}tp_string_;
```

An idea about the manipulation of lists would do no harm to our primary objective, the study of the working of the VM.

Consider two simple lists in python :

```
a = [1,2,3]
b = [4,5,6]
```

Again we start from tp_add which needs us to go back to our function.

The VM uses a function 'tp_extend' which returns a union that contains the new (extended) list.

You would see that the value of the field 'type' of the union that is returned would be 4 which indicates a list. Access the field 'list' as 'a.list'. 'list' is a structure variable that includes a pointer *val that points to another structure '_tp_list':

```
typedef struct _tp_list{
int gci;
tp_obj *items;
int len;
int alloc;
}_tp_list;
```

The pointer *items as you could see is of type tp_obj and deferencing it would give you the union tp_obj. This union contains a single element of the list.

Starting from the function tp_add:

```
*r.list.val -> items
```

would give you the union mentioned above.

Accessing the field 'val' of the field 'number' (variable of the structure tp_list_) of the union would give you the element of the list. Now , the obvious question is how do we obtain the next element of the list. For this you need to have some idea about the storage of unions .

```
r.list.val -> items
```

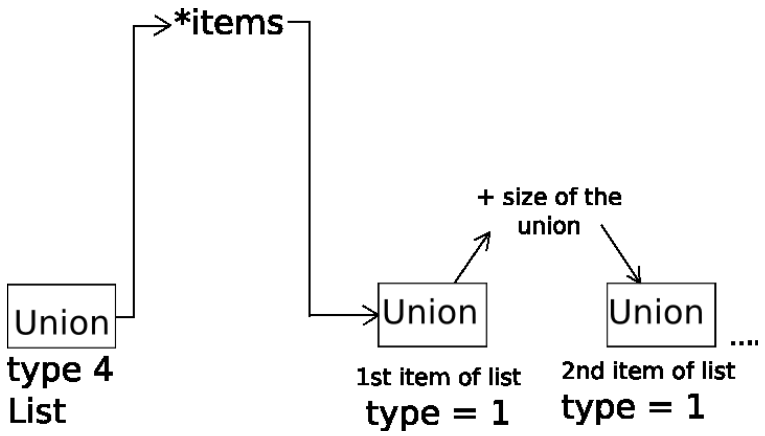
would give you the address of the union. Suppose the address be '0x96ca048' . Now to obtain the union which contains the next element of the list, you need to add the address of the current union with the size taken by each union (sizeof(union)).

Union containing the next element of the list = Address of the current union + size of each union. For example :

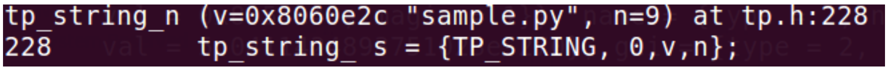
```
r.list.val -> items = 0x96ca048 + 10
```

would give you the union that stores the next element of the list .

The size of the union is 10, in hexadecimal = 16 bytes.

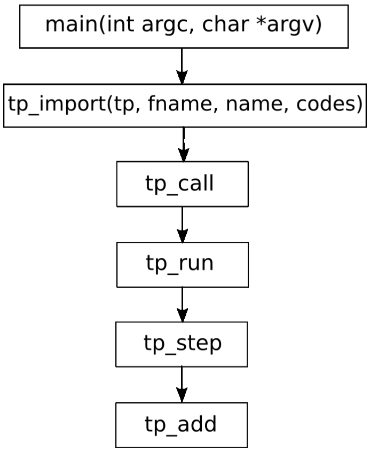

*items
+ size of the union
Union
type 4
List
Union
1st item of list
type = 1
Union
2nd item of list
type = 1
....

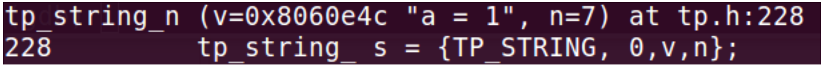
The Path of 'tp_add'



```
tp_string_n (v=0x8060e2c "sample.py", n=9) at tp.h:228
228         tp_string_ s = {TP_STRING, 0,v,n};
```

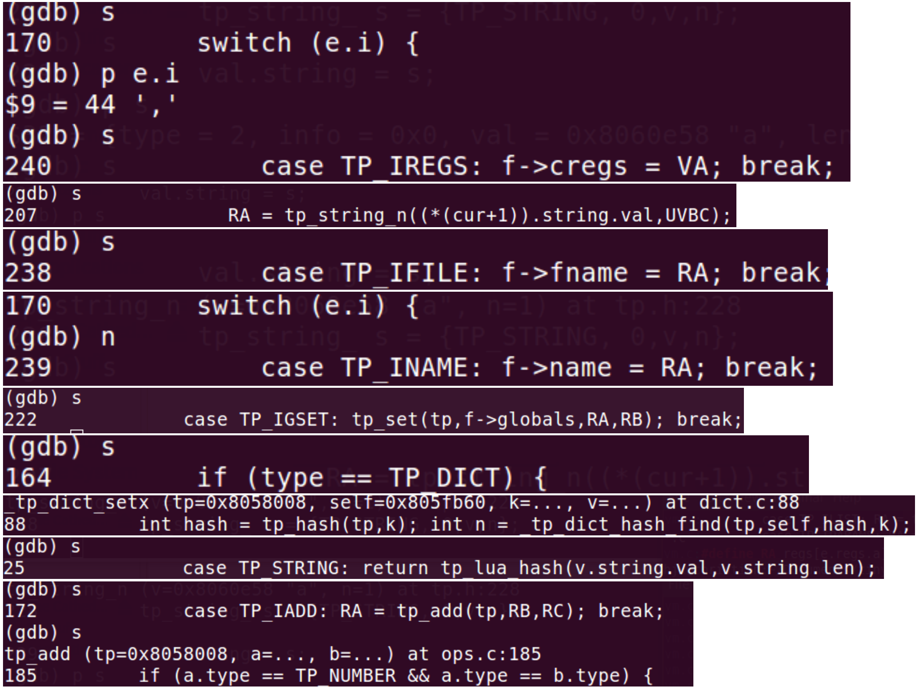
Back trace of the function calls which finally lead to execution of 'tp_add'



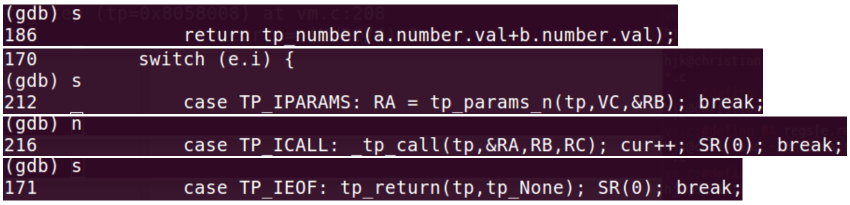


```
tp_string_n (v=0x8060e4c "a = 1", n=7) at tp.h:228
228         tp_string_ s = {TP_STRING, 0,v,n};
```

The switch () inside 'vm.c' is invoked nine times, one of which will be the selection of case: 'TP_IADD' and subsequent execution of 'tp_add'.



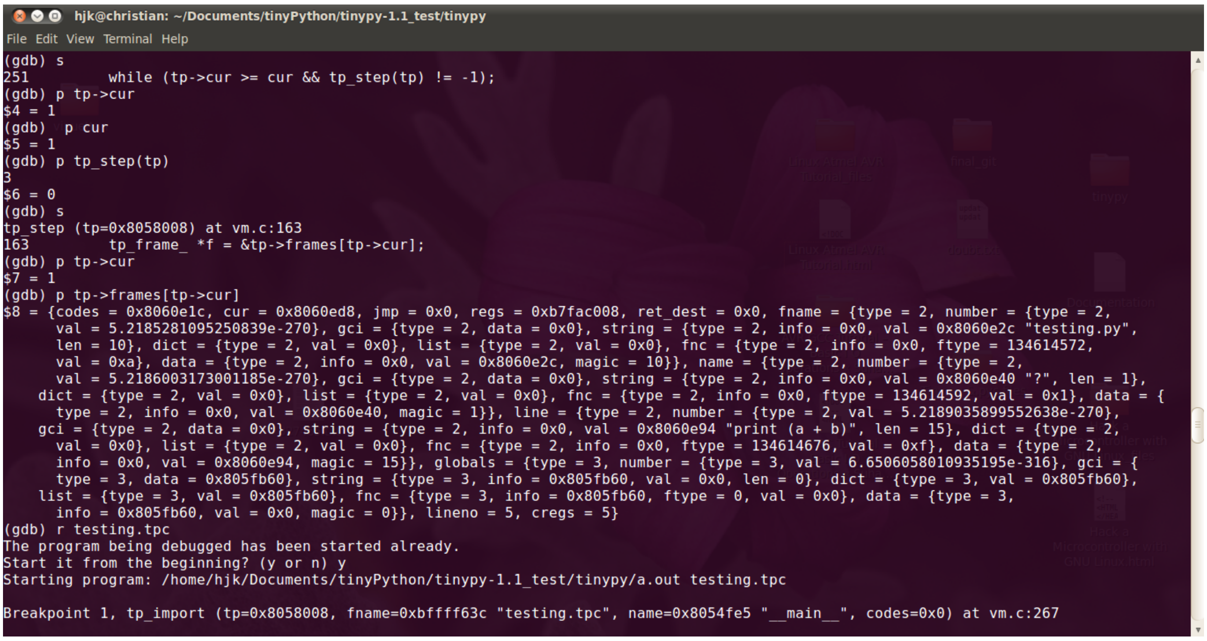
```
(gdb) s
170         switch (e.i) {
(gdb) p e.i
$9 = 44 ','
(gdb) s
240                 case TP_IREGS: f->cregs = VA; break;
(gdb) s
207                 RA = tp_string_n((*(cur+1)).string.val,UVBC);
(gdb) s
238                 case TP_IFILE: f->fname = RA; break;
170         switch (e.i) {
(gdb) n
239                 case TP_INAME: f->name = RA; break;
(gdb) s
222                 case TP_IGSET: tp_set(tp,f->globals,RA,RB); break;
(gdb) s
164         if (type == TP_DICT) {
_tp_dict_setx (tp=0x8058008, self=0x805fb60, k=..., v=...) at dict.c:88
88          int hash = tp_hash(tp,k); int n = _tp_dict_hash_find(tp,self,hash,k);
(gdb) s
25                  case TP_STRING: return tp_lua_hash(v.string.val,v.string.len);
(gdb) s
172                 case TP_IADD: RA = tp_add(tp,RB,RC); break;
(gdb) s
tp_add (tp=0x8058008, a=..., b=...) at ops.c:185
185         if (a.type == TP_NUMBER && a.type == b.type) {
```



```
(gdb) s
186             return tp_number(a.number.val+b.number.val);
170         switch (e.i) {
(gdb) s
212             case TP_IPARAMS: RA = tp_params_n(tp,VC,&RB); break;
(gdb) n
216             case TP_ICALL: _tp_call(tp,&RA,RB,RC); cur++; SR(0); break;
(gdb) s
171             case TP_IEOF: tp_return(tp,tp_None); SR(0); break;
```

A VM Snapshot

Another trial debugging was done with input file 'testing.tpc', whose source script file was 'testing.py' which just added two nmbers. Only the filename 'testing.tpc' was given to the debugger. The following snapshot shows the name of the source script 'testing.py', and the function 'print(a + b)' stored in fields of the VM data structures.


hjk@christian: ~/Documents/tinyPython/tinypy-1.1_test/tinypy
File Edit View Terminal Help

```
(gdb) s
251         while (tp->cur >= cur && tp_step(tp) != -1);
(gdb) p tp->cur
$4 = 1
(gdb)  p cur
$5 = 1
(gdb) p tp_step(tp)
3
$6 = 0
(gdb) s
tp_step (tp=0x8058008) at vm.c:163
163         tp_frame_ *f = &tp->frames[tp->cur];
(gdb) p tp->cur
$7 = 1
(gdb) p tp->frames[tp->cur]
$8 = {codes = 0x8060e1c, cur = 0x8060ed8, jmp = 0x0, regs = 0xb7fac008, ret_dest = 0x0, fname = {type = 2, number = {type = 2,
      val = 5.2185281095250839e-270}, gci = {type = 2, data = 0x0}, string = {type = 2, info = 0x0, val = 0x8060e2c "testing.py",
      len = 10}, dict = {type = 2, val = 0x0}, list = {type = 2, val = 0x0}, fnc = {type = 2, info = 0x0, ftype = 134614572,
      val = 0xa}, data = {type = 2, info = 0x0, val = 0x8060e2c, magic = 10}}, name = {type = 2, number = {type = 2,
      val = 5.2186003173001185e-270}, gci = {type = 2, data = 0x0}, string = {type = 2, info = 0x0, val = 0x8060e40 "?", len = 1},
    dict = {type = 2, val = 0x0}, list = {type = 2, val = 0x0}, fnc = {type = 2, info = 0x0, ftype = 134614592, val = 0x1}, data = {
      type = 2, info = 0x0, val = 0x8060e40, magic = 1}}, line = {type = 2, number = {type = 2, val = 5.2189035899552638e-270},
    gci = {type = 2, data = 0x0}, string = {type = 2, info = 0x0, val = 0x8060e94 "print (a + b)", len = 15}, dict = {type = 2,
      val = 0x0}, list = {type = 2, val = 0x0}, fnc = {type = 2, info = 0x0, ftype = 134614676, val = 0xf}, data = {type = 2,
      info = 0x0, val = 0x8060e94, magic = 15}}, globals = {type = 3, number = {type = 3, val = 6.6506058010935195e-316}, gci = {
      type = 3, data = 0x805fb60}, string = {type = 3, info = 0x805fb60, val = 0x0, len = 0}, dict = {type = 3, val = 0x805fb60},
    list = {type = 3, val = 0x805fb60}, fnc = {type = 3, info = 0x805fb60, ftype = 0, val = 0x0}, data = {type = 3,
      info = 0x805fb60, val = 0x0, magic = 0}}, lineno = 5, cregs = 5}
(gdb) r testing.tpc
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/hjk/Documents/tinyPython/tinypy-1.1_test/tinypy/a.out testing.tpc

Breakpoint 1, tp_import (tp=0x8058008, fname=0xbffff63c "testing.tpc", name=0x8054fe5 "__main__", codes=0x0) at vm.c:267
```