⇐ DEVELOPMENT-INDEX

The power of indirect threading code (ITC)

(c) Andreas Klimas 2017 (w3group, German)

With a tiny virtual engine which executes indirect threading code an application can be extended, tested and maintained at runtime by a simple programming language. The code could be stored even in a database so that we can get self executing data as well.

In this essay I wil show how ITC is working and how it can be implemented easily.

You can get the itc.c sample here (119 lines of C-code).

Concepts using in this essay

Execution Token (xt or word)	Is an entry in dictionary. It contains the name of the word (i.e. dup, print, sayHello, ..). If it is a high level word, xt contains an index into code_base where the word is defined. If the word is a special defining word, an additional index into code_base is maintained. Optional xt contains a source location from where this word is compiled from and optional a comment.

```
typedef struct xt_t {
   struct xt_t *next; // single linked list
   char *name; // name of this word
   int   data; // index into code_base for variables
               // or high level definitions
   int   does; // index into code_base for defining words
               // (i.e. create does>)
   char *location; // from where this word were compiled
} xt_t;
```

	For instrumenting (profiling, how many times this word were compiled, executed etc) this structure can be extended.
Dictionary	Is a single linked list or array with all known words (xt's). At runtime this list could be extended easily while loading some user extensions from database or files.
Code-Array (code_base)	While compiling or interpreting new definitions or variables are going to be placed into this array. If some memory is needed for the application, this could be taken from code_array as well.

Code-Index (here)	This is the first unused position of code_base. New definitions words, variables, are placed at this position and here will be advanced.
	`*here++=xt_lit; // compiling a literal which pushes`
	`                // "Hello World" on data stack`
	`*here++="Hello World"; // literal data (a string)`

Instruction-Index (ip, W)	While executing ip points to the next place in code_base which will be executed next.
	`xt_t **W;`
	`xt_t **ip=code_base;`
	`for(;;) {`
	`        W=*ip++;`
	`        (W->prim)();`
	`}`
	To be able to access the current executed xt at runtime, a global variable W is needed (not needed in OO languages where we have this or self).

Return-Stack-Index (rp)	This array maintains an index of return adresses for nested high level words.

Data-Stack-Index (sp)	This array maintains an index for arguments passed to the called word
Cell (cell_t)	cell_t is a union which is able to hold a pointer, integer, double, etc. Bit witdh of cell_t should be the same as *xt_t because we have a uniform data stack.

Summery of variables and types we need for a C based ITC implementation

```
typedef struct xt_t {
        struct xt_t *next;
        void (*prim)(void);// callback for primitive function
        int data; // index into code_base
        int ip; // index into code_base
} xt_t;
typedef union cell_t { // data stack type
        long long ival;
        char *cval;
        double *fval;
        xt_t *xt;
} cell_t;

#define MAX_CODE_SIZE 65536
#define MAX_STACK_DEPTH 256

xt_t   *dictionary; // linked list of words
xt_t   *code_base[MAX_CODE_SIZE];
int     here=0, ip=0;
int     rp_base[MAX_STACK_DEPTH];
int     rp=-1;
cell_t  sp_base[MAX_STACK_DEPTH];
int     sp=-1;
```

The virtual engine

As you see, the virtual engine for an ITC based system becomes very simple

```
xt_t *W; // current executed word
jmp_buf vm_halt;

static void f_halt(void) {longjmp(vm_halt, 1);}
void vm(void) {
        if(setjmp(vm_halt)) return; // VM halt primitive called

        for(;;) {
                W=code_array[ip++];
                (W->prim)();
        }
}
```

Compiling some code

In an other essay I show an implementation of an ITC interpreter and compiler, so I would show here only the concept of how the ITC is built with the minimal instruction set.

Firts we have to define some basic primitives

```
static xt_t *add_word(char *name, void (*prim)(void)) {
        xt_t *w=calloc(1, sizeof(xt_t));
        w->next=dictionary;
        dictionary=w;
        w->prim=prim;
        w->name=name;
        w->data=here;
        return w;
}
```

```
static void build_dictionary(void) {
   xt_halt=add_word("halt", f_halt); // halt virtual engine
   xt_exit=add_word("exit", f_exit); // return from word
   xt_docol=add_word("docol", f_docol); // enter high level word
   xt_lit=add_word("lit", f_lit); // literal
   xt_type=add_word("type", f_type); // print string on stdout
   xt_sub=add_word("-", f_sub); // subtract operation (top of stack
                                // from next of stack, leave result on
                                // top of stack
                                // ( a b -- a-b)  ( stackInput -- stackOutput)
   xt_while=add_word("(while)", f_while); // loop until top of stack is 0
   xt_drop=add_word("drop", f_drop); // drop top of stack
}
```

Then we are compiling a sample by hand. Note that for simplicity we are doing no code optimizing for now. With a simple optimizer this code becomes much smaller (peep hole optimizing, constant folding, tail call optimization).

```
int main() {
        build_dictionary();

        // Define high level word  hello
        xt_t *hello=add_word("hello", f_docol);
        code_base[here++]=xt_lit; // push Hello World on stack
        code_base[here++]=(void*)"Hello World\n";
        code_base[here++]=xt_type; // print Hello World
        code_base[here++]=xt_exit; // return from subroutine

        // Define high level word  foo
        xt_t *foo=add_word("foo", f_docol);
        code_base[here++]=xt_lit; // push down counter on stack
        code_base[here++]=(void*)10; // 10 times
        long long begin=here;           // this is our begin loop address
        code_base[here++]=hello;    // call the hello world
        code_base[here++]=xt_lit; // push decrement item
```

```
        code_base[here++]=(void*)1;
        code_base[here++]=xt_sub  ;// decrement by one
        code_base[here++]=xt_while;// repeat until top of stack becomes 0
        code_base[here++]=(void*)begin;
        code_base[here++]=xt_drop; // remove down counter
        code_base[here++]=xt_halt; // FINISHED, long jump

        ip=foo->data; // start of foo
        vm();
        return 0;
}
```

Makeing and running our vm

```
klimas@habibi:~/itc$ cc itc.c -o itc
klimas@habibi:~/itc$ ./itc
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
klimas@habibi:~/itc$
```

Changing vm() to print each instruction executed

```
        for(;;) {
                W=code_base[ip++];
                printf("vm:%s\n", W->name);
```

```
                (W->prim)();
        }
```

Leads to this output (only first loop displayed here)

```
vm:lit
vm:bar
vm:lit
vm:type
Hello World
vm:exit
vm:lit
vm:-
vm:(while)
vm:drop
vm:halt
```