Hacking STklos

Jeronimo Pellegrini

Table of Contents

This is a quick guide to STklos hacking. It's not detailed, so the document doesn't become huge, and also because after understanding the basics, hacking STklos should not be difficult.

Chapter 1. Directories

The subdirectories in the STklos source tree are:

- `doc` – documentation, written mostly in Skribe
- `etc` – various sample files for specific needs
- `examples` – examples (oh, who could tell?)
- `ffi` – `libffi` (a local copy)
- `gc` – the Boehm-Demers-Weiser garbage collector, libgc (a local copy)
- `gmp` – a slow compatible GNU MP)
- `lib` – Scheme files, including from basic things like the boot program up to high-level things like modules implementing SRFIs
- `pcre` – `libpcre` (a local copy)
- `pkgman` – the package manager
- `src` – the STklos core, written in C
- `tests` – the tests, of course!
- `utils` – utilities and wrappers

Chapter 2. STklos initialization

`main` is in `src/stklos.c`, where command line options are parsed and the scheme interpreter is started:

- `STk_init_library` – performs library initialization. This is done in `src/lib.c`, which is a very simple file that just calls several initialization functions. Those functions are defined in different files under `src/`;
- `build_scheme_args` – collects the command line options in the variable **`%program-args`**;
- `STk_load_boot` – loads the boot file (if one is to be loaded);
- `STk_boot_from_C` – actually boots the Scheme interpreter. This function is defined in `src/vm.c`, where the STklos virtual machine code is.

In order to include Scheme code for execution during STklos startup, edit `lib/boot.stk`.

Chapter 3. Adding simple modules and SRFIs

3.1. Adding modules

- add your `fantastic-module.stk` to `lib/`
- include `fantastic-module.stk` and `fantastic-module.ostk` in the variables `SRC_STK` and `scheme_OBJS`, in `lib/Makefile.am`
- Tests reside in the `tests` directory. Create a new file in `tests` directory and include it in the list of loaded files in `do-test.stk`

3.2. Adding SRFIs

In order to add SRFI 9999 to STklos,

- add your `9999.stk` to `lib/srfi`
- include `9999.stk` and `9999.ostk` in the variables `SRC_STK` and `SRC_OSTK`, in `lib/srfi/Makefile.am`
- Add a line describing it in `lib/srfis.stk` (the format is described in the file itself).
- Tests reside in the `tests` directory. Add the tests in a file `tests/srfis/9999.stk`

For new SRFIs, adding its description in `lib/srfis.stk` suffices to update

- the `SUPPORTED-SRFIS` in the main directory
- launch the tests you added in `tests/srfis` directory, and
- add an automatically generated documentation for this SRFI

3.3. Mixed SRFIs (Scheme and C)

To add a mixed SRFI 8888,

- Write a `8888.c` file and put it in `lib/srfi`
- Write a `8888.stk` Scheme file and also put it in `lib/srfi`
- Add your mixed SRFI to `lib/srfi/Makefile.am`, in the section `` `SRFIs written in C and Scheme'' (variables `SRC_C``, `SRC_C_STK`, and `SRC_SHOBJ`

3.3.1. Content of the Scheme file

The Scheme file will be compiled as a byte-code stream embedded in C. Here, the compiled file will be called `$DIR/srfi-170-incl.c`. It is built by the `utils/tmpcomp` script with

```
../../utils/tmpcomp -o srfi-170-incl.c $DIR/srfi-170.stk
```

Note: when the destination file ends with a `.c` suffix, the `tmpcomp` command produces a C file instead of a byte-code file.

You don't have to pay attention to any particular point in the writing of this file.

3.4. Content of the C file

The C file must follow the conventions of dynamically loadable code as shown in the example in the `/etc` directory.

In this C file, to use the previously compiled Scheme code, you have to (using SRFI 170 as an example):

- include the file `170-incl.c` at the top of your C file
- add a call to execute the Scheme code just before the `MODULE_ENTRY_END` directive. This is done with the following invocation:

```
    STk_execute_C_bytecode(__module_consts, __module_code);
```

- Add a directive `DEFINE_MODULE_INFO` at the end of the file. It permits to access some information of the module (STklos version used to compile the module, exported symbols, ...). For now, this information is not used, but omitting to add this directive will probably lead to a compiler warning about an unresolved reference.

As one more example, SRFI 25 has, at the end of the C file:

```
MODULE_ENTRY_START("srfi/25")
{
  SCM module =  STk_create_module(STk_intern("srfi/25"));
  STk_export_all_symbols(module);

  ADD_PRIMITIVE_IN_MODULE(...);
  ...
  ...

  /* Execute Scheme code */
  STk_execute_C_bytecode(__module_consts, __module_code);
}
MODULE_ENTRY_END

DEFINE_MODULE_INFO
```

See SRFI-25, SRFI-27 and SRFI-170 as a reference.

3.5. Documentation

3.5.1. Documenting SRFIs in `srfi.skb`

General documentation is automatically generated for SRFIs. If you need to give a precision specific to a given SRFI, add it to the end of the `doc/skb/srfi.skb` file using the `gen-srfi-documentation` function.

Note that the documentation is written in Skribe tool which is no more maintained. Consequently, the documentation will not be generated. The HTML and PDF documentation is rebuilt from time to time by @egallesio.

3.5.2. Documenting primitives written in C

Before `DEFINE_PRIMITIVE`, add a comment similar to the others you see in the C files. An example:

```
/*
<doc EXT bignum?
 * (bignum? x)
 *
 * This predicates returns |#t| if |x| is an integer number too large to be
 * represented with a native integer.
 * @lisp
 * (bignum? (expt 2 300))     => |#t|   (very likely)
 * (bignum? 12)               => |#f|
 * (bignum? "no")             => |#f|
 * @end lisp
doc>
*/
DEFINE_PRIMITIVE("bignum?", bignump, subr1, (SCM x))
{
  return MAKE_BOOLEAN(BIGNUMP(x));
}
```

Pay attention to the parts of this comment: it begins with the primitive name, then there's an explanation, then examples in Scheme. Wrap symbols/identifiers in `|.|`; use `@lisp` and `@end lisp@` to show an example of usage.

Chapter 4. Writing primitives in C

Use the macro DEFINE_PRIMITIVE:

```
DEFINE_PRIMITIVE("fixnum?", fixnump, subr1, (SCM obj))
{
  return MAKE_BOOLEAN(INTP(obj));
}
```

The arguments for this example are

- Scheme name
- C function name (its full name will have the string ```STk_'' prepended to it)
- the type of primitive (in this case, it is a subroutine with one parameter – ```subr1''
- the arguents, surrounded by parentheses. In this case there is only one argument, `obj', and its type is `SCM'' (which is the type of all Scheme objects in STklos).

Then add it:

```
ADD_PRIMITIVE(fixnump);
```

The name passed to ADD_PRIMITIVE is the C function name.

4.1. Calling Scheme primitives

Recall that a primitive is defined like this:

```
DEFINE_PRIMITIVE("fixnum?", fixnump, subr1, (SCM obj))
{ ... }

ADD_PRIMITIVE(fixnump);
```

TO use this primitive later in C code, add the STk_ prefix to its C function name:

```
if (STk_fixnump(obj) == STk_false) ...
```

4.2. Returning values

`STk_n_values(n, v1, v2, ⋯, vn)` returns `n` values from a procedure.

For example, `read-line` (defined in `port.c`) has these two lines:

```
return STk_n_values(2, res, STk_eof)
```

for when it found the end of the file, and

```
return STk_n_values(2, res, delim);
```

for when it did not yet reach EOF, so it returns the line delimiter as second value.

4.3. Errors

The C function that raises errors is

- `STk_error(fmt, arg1, arg2, ⋯)` – the STklos error procedure. `fmt` is a format string, and after it there are arguments.

But as you can see in the top of several C files, it is useful to define wrappers:

```
static void error_bad_number(SCM n)
{
  STk_error("~S is a bad number", n);
}

static void error_at_least_1(void)
{
  STk_error("expects at least one argument");
}

static void error_cannot_operate(char *operation, SCM o1, SCM o2)
{
  STk_error("cannot perform %s on ~S and ~S", operation, o1, o2);
}
```

4.4. Unboxed types

The trditional way to representa data in Lisp languages is by *tagged objects*. A long enough machine word is used to represent all types, and some bits are reserved to distinguish the type of the object. In STklos, the *two least significant bits* are used for this.

- `00` - pointer on an object descriptor (a box)

- `01` - fixnum
- `10` - small object (characters and others)
- `11` - small constant (`#t`, `#f`, `'()`, `#eof`, `#void`, dot, close-parenthesis)

The idea is that checking the type of these should be very fast, because it is done at runtime, so to check wether an object is `#eof`, one needs only check if `obj & 0x4 == 0x3` (but usually, we have macros for that).

STklos uses C `long` words so, for example, in a machine where `long int` is 32 bits long the bit sequence

```
0000 0000 0000 0000 0000 0000 0010 0101
```

is a *fixnum* (because its two least significant digits are `01`, and the value of the fixnum is 9 (because after discarding the `01` that is on the right of the sequence, the number left is `1001`).

4.4.1. Booleans

- `STk_true` is the SCM object for `#t`
- `STk_false` is the SCM object for `#f`
- `BOOLEANP(o)` checks wether the object `o` is boolean (the macro actually does `(o) == STk_true) || ((o) == STk_false`
- `MAKE_BOOLEAN(_cond)` expands to a conditional statement: if `_cond` is true, then the value is `STk_true`, otherwise it is `STk_false`.

4.4.2. Fixnums

Fixnums are not allocated but have their two least significant bits set to `01` (in Lisp-parlance, it has `01` as its *tag*).

- `INTP(o)` - returns STklos_true if `o` is a Scheme integer or `STklos_false` otherwise
- `MAKE_INT(n)` - takes a `long` C number and turns it into an `SCM` integer object. Actually, this will shift the number to the left by two positions and insert the tag If we could represent numbers as binary in C, it would be like this:

```
MAKE_INT( 000011000 )  // --> 001100001
```

- `INT_VAL(o)` - returns the value of the fixnum `o`, as a C `long` value (the opposite of the previous operation)

4.5. Boxed types

Boxed types are anything except for fixnums, small objects and small constants. They are tagged with `00`.

- `BOXED_OBJP(o)` – true if `o` is a boxed object
- `BOXED_TYPE_EQ(o,t)` – checks wether `o` is a boxed object of type `t`
- `BOXED_TYPE(o)` – returns the type of boxed object `o`
- `BOXED_INFO` – returns the information of boxed object `o`

The type definition for all possible types, in `stklos.h`, is self-explanatory:

```
typedef enum {
  tc_not_boxed=-1,
  tc_cons, tc_integer, tc_real, tc_bignum,  tc_rational,                /* 0 */
  tc_complex, tc_symbol, tc_keyword, tc_string, tc_module,              /* 5 */
  tc_instance, tc_closure, tc_subr0, tc_subr1, tc_subr2,                /* 10 */
  tc_subr3, tc_subr4, tc_subr5, tc_subr01, tc_subr12,                   /* 15 */
  tc_subr23, tc_vsubr, tc_apply, tc_vector, tc_uvector,                 /* 20 */
  tc_hash_table, tc_port, tc_frame, tc_next_method, tc_promise,         /* 25 */
  tc_regexp, tc_process, tc_continuation, tc_values, tc_parameter,      /* 30 */
  tc_socket, tc_struct_type, tc_struct, tc_thread, tc_mutex,            /* 35 */
  tc_condv, tc_box, tc_ext_func, tc_pointer, tc_callback,               /* 40 */
  tc_last_standard /* must be last as indicated by its name */
} type_cell;
```

4.5.1. Lists

Here are some primitives for lists, for example:

- `CAR(p)` – equivalent to Scheme `car`: returns the car of `p` (an SCM object)
- `CDR(p)` – equivalent to Scheme `cdr`: returns the car of `p` (an SCM object, which certainly is a list)
- `CONSP(p)` - equivalent to Scheme `cons?`
- `NULLP(p)` - equivalent to Scheme `null?`
- `STk_cons` - equivalent to Scheme `cons`

4.5.2. Strings

Another example are strings. They are defined as the following structure:

```
struct string_obj {
  stk_header header;
  int space;            /* allocated size  */
  int size;             /* # of bytes used */
  int length;           /* "external" length of the string */
  char *chars;
};
```

Then, some primitives:

```
#define STRING_SPACE(p)  (((struct string_obj *) (p))->space)
#define STRING_SIZE(p)   (((struct string_obj *) (p))->size)
#define STRING_LENGTH(p) (((struct string_obj *) (p))->length)
#define STRING_CHARS(p)  (((struct string_obj *) (p))->chars)
#define STRINGP(p)       (BOXED_TYPE_EQ((p), tc_string))
```

The following primitives are defined in a `str.c`, but `stklos.h` is used by several files use them, so they're included with `EXTERN_PRIMITIVE`:

```
EXTERN_PRIMITIVE("string=?", streq, subr2, (SCM s1, SCM s2));
EXTERN_PRIMITIVE("string-ref", string_ref, subr2, (SCM str, SCM index));
EXTERN_PRIMITIVE("string-set!", string_set, subr3, (SCM str, SCM index, SCM value));
EXTERN_PRIMITIVE("string-downcase!", string_ddowncase, vsubr, (int argc, SCM *argv));
```

4.6. Dynamically loadable modules

See some examples in `etc/`

4.7. Creating new types

We'll be using SRFI-25 as an example. In that SRFI, am `array` type is created.

- Create a C struct whose first field is of type `stk_header`

```
struct array_obj {
  stk_header header;
  int shared;                /* does this array share data with another? */
  int *orig_share_count;     /* pointer to original array share counter */
#ifndef THREADS_NONE
  MUT_FIELD(share_cnt_lock); /* lock for share counter */
  MUT_FIELD(*share_cnt_lock_addr); /* pointer to mutex - ours or of original array's
*/
#endif
  long size;                 /* size of data */
  long length;               /* # of elements */
  int  rank;                 /* # of dimensons */
  long offset;               /* offset from zero, to be added when calculaing index */
  long *shape;               /* pairs of bounds for each dimenson */
  long *multipliers;         /* size of each dimension stride */
  SCM  *data_ptr;            /* pointer to data */
};
```

The fields in the struct may contain both C and Scheme elements (the Scheme elements have `SCM` types).

- Maybe create some accessor macros

```
#define ARRAYP(p)           (BOXED_TYPE_EQ((p), tc_array))
#define ARRAY_SHARED(p)     (((struct array_obj *) (p))->shared)
#define ARRAY_SHARE_COUNT(p) (((struct array_obj *) (p))->orig_share_count)
#define ARRAY_LOCK(p)       (*((struct array_obj *) (p))->share_cnt_lock_addr))
#define ARRAY_SIZE(p)       (((struct array_obj *) (p))->size)
#define ARRAY_LENGTH(p)     (((struct array_obj *) (p))->length)
#define ARRAY_RANK(p)       (((struct array_obj *) (p))->rank)
#define ARRAY_OFFSET(p)     (((struct array_obj *) (p))->offset)
#define ARRAY_SHAPE(p)      (((struct array_obj *) (p))->shape)
#define ARRAY_MULTS(p)      (((struct array_obj *) (p))->multipliers)
#define ARRAY_DATA(p)       (((struct array_obj *) (p))->data_ptr)
```

Be mindful of thread-related things: not all STklos builds have threading enabled!

```
#ifdef THREADS_NONE
#  define ARRAY_MUTEX(p)
#  define ARRAY_MUTEX_SIZE 1
#else
#  define ARRAY_MUTEX(p) (((struct array_obj *) (p))->share_cnt_lock)
#  define ARRAY_MUTEX_SIZE (sizeof(pthread_mutex_t))
#  define ARRAY_MUTEX_PTR_SIZE (sizeof(pthread_mutex_t*))
#endif
```

- Create an extended type descriptor which contains the type name, and pointers to functions to print and compare elements:

```
static void print_array(SCM array, SCM port, int mode)
{
  /*
    Here goes the code for printing array.
    Use the functions
      - STk_puts(char *str, SCM port)
      - STk_print(SCM obj, SCM port, int mode)
    It may be useful to first create a buffer, use snprintf on it, then
    use STk_puts to print it.
   */
}
```

```
static SCM test_equal_array(SCM x, SCM y)
{
 /*
   Code that retruns STk_true if x and y are to be considered `equal?`,
   and STk_false othereise.

   NOTE: remember to NOT return 0 or 1. THe return value should be a Scheme
         object, not a C with the intended boolean value. This is particularly
         important because the compiler will NOT warn you if you return "0":
         - SCM is defined as a pointer to void
         - '0' can be interpreted as a pointer, so the compiler thinks it's OK
         - '0' is *not* the same as STk_void
  */
}
```

```
static struct extended_type_descr xtype_array = {
  .name  = "array",
  .print = print_array,
  .equal = test_equal_array
};
```

- At the end of your C code, inside the MODULE_ENTRY_START part, initialize an element of the new type: `tc_array = STk_new_user_type(&xtype_array);`
- Create a describing procedure:

```
(%user-type-proc-set! 'array 'describe
                      (lambda (x port)
                        (format port "an array of rank ~A and size ~A"
                                (array-rank x)
                                (array-size x))))
```

- Define a class, and associate it with the type name you have created.

```
(define-class <array> (<top>) ())
(export <array>)

(%user-type-proc-set! 'array 'class-of <array>)
```

- If objects of the new type will have a printed representation, create a reader procedure:

```
(define-reader-ctor '<array>
  (lambda args
    (apply array (apply shape (car args)) (cdr args))))
```

Chapter 5. The virtual machine

See the file `vm.adoc` for a description of the opcodes.

Chapter 6. Compiler and optimizations

6.1. The compiler

The compiler is in the file `lib/compiler.stk`.

There is a `compile` procedure at the end of the file, whose logic is very simple:

1. expand macros
2. compile special forms
3. if what's left is a symbol, compile a call
4. if it's not a symbol, compile it as a constant

In the rest of the file, there are procedures to compile different special forms and inlinable primitives.

The code is generated as a list, in the **`code-instr`** global variable in the `STKLOS-COMPILER` module. The procedure `emit` conses one more instruction on the code (which will later be reversed, of course)

6.2. Peephole optimizer

STklos uses a peephole optimzier, located in the file `lib/peephole.stk`. This optimizer will transform several instruction patterns in the generated code into more efficient ones. For example:

```
;; [SMALL-INT, PUSH] => INT-PUSH
((and (eq? i1 'SMALL-INT) (eq? i2 'PUSH))
 (replace-2-instr code (list 'INT-PUSH (this-arg1 code))))
```

This transforms two instructions (`load a small integer into `val', then push it onto the stack') into one single instruction (push an integer onto the stack).

The peephole optimizer also reduces the size of the bytecode:

```
;; [RETURN; RETURN] => [RETURN]
((and (eq? i1 'RETURN) (eq? i2 'RETURN))
 (replace-2-instr code (list 'RETURN)))
```

This will turn two adjacent `RETURN` instructions into a single one, making the object file smaller. This is valid because there won't be any `GOTO` pointing to the second instruction; if this was the case, then

the code would have a label between the two `RETURN`s.

Another example is `GOTO` optimization:

```
;; [GOTO x], ... ,x: GOTO y => GOTO y
;; [GOTO x], ... ,x: RETURN => RETURN
((eq? i1 'GOTO)
 (set! code (optimize-goto code)))
```

The procedure `optimize-goto-code`, also in the file `peephole.stk`, will perform the transformations indicated in the comments.

The input code is represented as a list. Some relevant definitions are in the beginning of the file:

```
(label? code)      ; is thecurrent instruction a label?
(this-instr code)  ; the current instruction (reference to a position in the list)
(next-instr code)  ; the next instruction (cdr of the current one)
(this-arg1 code)   ; argument 1 of current instruction
(this-arg2 code)   ; argument 2 of current instruction
(next-arg1 code)   ; argument 1 of next instruction
(next-arg2 code)   ; argument 2 of next instruction
```

Chapter 7. Garbage collection

STklos uses the Boehm-Demers-Weiser garbage collector. The wrapper for the GC is located in the header file `src/stklos.h`:

```
#define STk_must_malloc(size)           GC_MALLOC(size)
#define STk_must_malloc_atomic(size)    GC_MALLOC_ATOMIC(size)
#define STk_must_realloc(ptr, size)     GC_REALLOC((ptr), (size))
#define STk_free(ptr)                   GC_FREE(ptr)
#define STk_register_finalizer(ptr, f)  GC_REGISTER_FINALIZER( \
                                            (void *) (ptr),             \
                                            (GC_finalization_proc)(f),  \
                                            0, 0, 0)
#define STk_gc()                        GC_gcollect()

void STk_gc_init(void);
```

- `STk_must_malloc` - used to allocate structured objects.
- `STk_must_malloc_atomic` - used when there won't be any pointers inside the object, and we don't want to confuse the GC with patterns that are supposed to be just a bignum, but ``look like apointer''. Used for strings, numbers etc.
- `STk_register_finalizer` will register a finalizer function `f`, which will be called when the object at `ptr` is collected.