8 Translation from tySECK to k86

We need to implement the semantics of tySECK codes by k86 intructions. That is, we need to implement the three components of tySECK machine (stack S, environment E, and continuation K) and we need to implement the tySECK values in k86 objects.

- S, E, and K are implemented as stacks in separate memory segments of k86 machine.
- Primitive values in tySECK are directly implemented as primitive values in k86, except that boolean values are implemented in k86 as integers: true as non-zero integers and false as zero.
- Function values (C, E) is implemented as a pointer to a pair whose first field has the code label and the second field the pointer to the environment. Similarly for continuation pair (E, C). The code label is the label at the beginning of the code.
- Now the question: how and where do we implement the function's environment part? Can the environment be implemented as a pointer to the current machine environment E? No, because machine environment and function value's environment have different life-spans. The environment when a function is called is in general different from the environment which was captured when the function is defined. The values in the current machine environment E can be expired (hence the memories that were used to implement the values are re-cycled) while the captured environment in functions can last. For example, consider the following program:

  ```
  let
   f = let x = 2 in fn y => x+y end              (1)
  in
   f z                                           (2)
  end
  ```

  The `f` has a function value whose environment part has the entry [`x` $\mapsto$ 2]. This entry in the machine environment is removed (by the `DEFRAME` command) when the `let`-block at line (1) is exited. Note that this entry has to be available when the function is called at line (2).
- Thus, function's environment has to be implemented in a separate memory area whose objects last "forever" without being expired. We call this area "heap."

We now know that four memory areas are necessary:

```
AREA S  // stack
AREA E  // environment
AREA K  // continuation
AREA H  // heap
```

Each area's first empty address is pointed to by each pointer:

```
AREA sp // stack pointer sp
AREA ep // environment pointer ep
AREA kp // continuation pointer kp
AREA hp // heap pointer hp
```

In defining the translation rule we use the following macros:

```
macro push(x, ptr)                      =  MOVE x ptr@
                                           ADD 1 ptr@ ptr
macro pop(ptr)                          =  SUB ptr@ 1 ptr
macro copy(REPEAT, from, to, n, END)    =  LAB REPEAT
                                            JMPZ n@ END
                                            JMPN n@ END
                                            MOVE from(n@)@ to(n@)
                                            SUB n@ 1 n
                                            JMP REPEAT
                                           LAB END
```

The translation function `codegen` is of type

```
codegen:  tyCmd -> k86Cmd * k86Cmd
```

The second output is for the k86 codes of functions that appear in the given tySECK command. Given a sequence of tySECK command C, `codegen`(C) will return two k86 code-sequences

$$(C_1,\ C_2)$$

from which the final k86 code would be

$$\texttt{LAB START}\ C_1\ \texttt{LAB END}\ C_2$$

with appropriate area declarations in front of the `START` label.

The `codegen` function would be defined something like in Figure 7. In the figure I omitted the second part when it's empty.

```
fun codegen FRAME =  push(sp@(-1),ep)
                     pop(sp)
  | codegen DEFRAME  = pop(ep)
  | codegen FETCH(n) = push(ep@(-n-1),sp)
  | codegen QUOTE(n) = push(n,sp)
  | codegen QUOTE(s) = push(s,sp)
  | codegen QUOTE(true) = push(1,sp)
  | codegen QUOTE(false) = push(0,sp)
  | codegen ADD =  ADD sp@(-1) sp@(-2) sp@(-2)
                   SUB sp@ 1 sp
  | codegen COND(C1, C2) =
      let
         val C1 = codegen C1
         val C2 = codegen C2
         val ELSE = newLabel()
         val EXIT = newLabel()
      in
            JMPZ sp@(-1) ELSE
            pop(sp)
            C1
            JMP EXIT
        LAB ELSE
            pop(sp)
            C2
        LAB EXIT
      end
  | codegen FTN(C) =
      let
         val C = codegen C
         val ENTRY = newLabel()
         val REPEAT = newLabel()
         val END = newLabel()
      in
       ( TOZ ep@ sp@           (* sp@@ has the current environment size *)
         push(sp@@, hp)        (* put the environment size at the head of
                                  the copied environment *)
         copy(REPEAT, ep@, hp@, sp@, END) (* copy the env into the heap *)
         ADD hp@ sp@@ hp       (* shift the hp@ by the env size *)
         ADD hp@ 2 hp          (* allocate the pair cell in the heap *)
         MOVE ENTRY hp@(-1)    (* second field has the code label *)
         MOVE hp@(-(sp@+3))
              hp@(-2)          (* first field has the ptr to the copied env *)
         push(hp@(-2),sp)      (* stack has the pointer to the pair *)
       , C)
      end
  | ...
```

그림 7: tySECK-to-k86 Translation Rules

8.1 Economic/Optimal Translation into k86: Challenges and Opportunities

The translation procedure in the previous section has a lot room for improvement.

- When a function value is made, taking one copy of the current environment into the heap area takes a long time. Under some conditions we don't have to copy the environment. The current environment can be just pointed to. What would be the conditions?

- When a function is called, the return state (the pair of the current environment and next command to continue after returning) is pushed into the continuation stack K. In particular, the environment part is copied. Under some conditions we can avoid this expensive copy process, too. What would be the conditions?

- When a recursive function is unrolled by the `UNROLL` command, the environment part E' is duplicated:

  $$(S,\, v.E',\, \underbrace{\texttt{UNROLL(}rec\,\tau_1 \to \tau_2\texttt{,}\ \ C'\texttt{)}.C}_{C''},\, K) \to (S,\, v.(C'', E').E',\, C,\, K)$$

  I guess we can always avoid this duplicated copy. Do you agree?

Other than the above big chances for optimizations we can do many simple optimizations after the translation is done by examining a small number of tySECK or k86 codes in sequence. To enlist some,

- When we see a tySECK code sequence like

  ```
  QUOTE(1).QUOTE(2).ADD
  ```

  we can replace it by

  ```
  QUOTE(3)
  ```

- Two instructions `move` m_1 `sp@` immediately followed by `move sp@@` m_2 can be replaced by a single command `move` m_1 m_2.