Slip - Coroutines

Milan Lagae

Institution/University Name:	Vrije Universiteit Brussel
Faculty:	Sciences and Bioengineering Sciences
Course:	Programming Language Engineering

Contents

1 Intro

This report will discuss an implementation for the assignment "Exam assignment: Coroutines" for the course:"Programming Language Engineering".

First, the implementation proceeding the final one is shown in section §3 and reasoning, followed by the final implement in section §4.

2 Slip version

The Slip version used to implement the assignment, is version 9. The following list of files were modified: SlipCompile.*, SlipEvaluate.*, SlipGrammar.*, SlipThread.*, SlipMain.*, SlipNative.*.

3 First iteration

This subsection will discuss the initial implementation, before arriving at the final implementation.

3.1 Newprocess

For the first iteration, the runtime type used for storing information required for context switching was the PRC_type.

On evaluation of the newprocess, the make_coroutine procedure uses the make_PRC to create a procedure (lambda) named as the name given to the newprocess.

The PRC_type runtime type has a env field, to which the Context field was stored by casting the CNT_Type to a VEC_type, as required by the make_PRC function.

The Context value being the result of calling Thread_Pop after pushing Continue_newprocess_body on top of the stack using Thread_Push. The pushed continuation being of the neP type, consisting of the body, body_size and procedure name.

3.2 Transfer

When entering the transfer_native, the runtime expression is checked for the correct runtime type, in this case being the PRC_type.

If both arguments are of the PRC_type, the execution continues. Before context switch can take place. The env field on the PRC_type is checked for a value, in this case the CNT_type, which is the Context value from earlier.

If that is the case, the context switch can take place, the env field is cast to CNT_type and the Thread_Replace method is called with the value to_context as input. The value Main_Unspecified is returned.

The program will continue with the current thread on the stack, which is the Continue_newprocess_body continuation.

The interpreter will enter the procedure: continue_newprocess_body. The current thread value is retrieved using Thread_Peak. From the current thread, the values: body, bsz are retrieved. The body is evaluated using the evaluate_inline_sequence procedure.

3.3 Problems

Running the above implementation with specific experiment files mentioned in §5, showcases some shortcomings in the implementation.

Executing the script: roundrobin-bug.slip, showcases a bug with switching the environments between the processes, as shown in Figure 1.

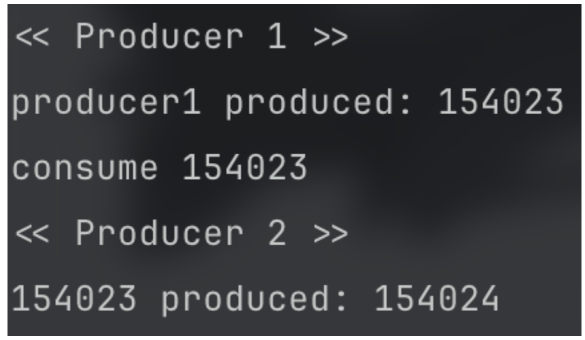

Figure 1: Round robin Environment bug

A similar issue occurs when executing the ping-pong2.slip experiment, the output is continuously ping instead of the expected ping, pong ... output.

4 Design Choices

This section will discuss the choices made in the final implementation.

4.1 Grammar

The PRC_type runtime type is replaced with the COR_type. The C0R_type consists of the following fields:

- name, name of the coroutine;
- cnt, continuation of the coroutine;
- env, storing the environment at the moment of transfer;
- frm, storing the frame at the moment of transfer.

As other grammar types, grammar tags and auxiliary procedures are also created. The NEP_type is the same as in the §3 implementation. The NEP_type can be considered the compile type while COR_type the runtime type.

Since in contrast to other special forms, the information required during the compilation & evaluation steps is different.

4.2 Compilation

During the compilation step, there is no need to push the list of Operands on the stack, because in contrast to while or if, there is no compile processes that needs to take place between retrieving the name & compiling the body of the process.

To be absolutely sure, the Operands value is still claimed. Following the compilation of the body, the output is claimed to prevent any garbage collection.

4.3 Evaluation

The runtime evaluation that created the PRC_type is replaced with the procedure call as shown in Listing 1.

```
// Create coroutine with its process context
coroutine = make_coroutine(name, process_stack);
```

Listing 1: Evaluation - Make coroutine

Less information is now used for creating the runtime type COR_type, consisting of the name and process_stack. The latter being the CNT_type thread value.

The make_coroutine procedure, creates a COR_type procedure using the grammar method: make_COR as shown in Listing 2.

```
static COR_type make_coroutine(EXP_type Name, CNT_type Context) {
  COR_type coroutine;
  VEC_type environment, frame;

  // Get the environment & frame
  environment = Environment_Get_Environment();
  frame = Environment_Get_Frame();

  // Set the name, context (CNT), environment and frame of the coroutine
  coroutine = make_COR(Name, Context, environment, frame);

  // Return the coroutine procedure
  return coroutine;
}
```

Listing 2: Evaluation - Make coroutine

The env and frm values of the COR_type are set as the environment & frame at moment of evaluation.

4.4 Transfer

The expected runtime values are now of the COR_type type. Once the grammar tags are confirmed for both values, the execution may continue. The context check is now performed on the cnt field instead of on the env field.

The current continuation state is saved in the cnt field of the from_process, in addition so are the current state of the environment and frame values.

With the state of the from_process saved, the context switch transfer can continue. The environment (env) and frame (frm) values are retrieved from the process.

The current environment & frame are replaced with the previously retrieved values.

The last step to complete the transfer is setting the currently active thread. This is done using Thread_Replace with the to_context continuation as value.

The transfer function itself does not return a value (Main_Unspecified). The continuation stack will proceed to execute the first thread on the stack, which contains a continuation to the function: continue_newprocess_body.

4.5 Problems

The bug alluded to earlier in Figure 1 has now been fixed, and the output proceeds as expected.

The current implementation, for the ping-pong2.slip experiment will output the expected output, but after a few seconds the error: insufficient memory will appear. At the moment of writing no fix for this issue has been found.

The output for the call-rely.slip experiment also does not match the expected output. The program terminates after 3 iterations.

5 Experiments

The complete list of experiments can be found in the slip directory. The following list of experiments illustrating the coroutines are included:

1. single-process.slip
2. ping-pong.slip
3. ping-pong2.slip
4. producer-consumer.slip
5. call-reply.slip
6. roundrobin.slip
7. roundrobin-bug.slip
8. circle.slip

Majority of the examples have been slightly modified to work within the constraints of the current implementation.

5.1 Examples

The experiments: ping-pong.slip, producer-consumer.slip and call-reply.slip are the examples described in the project assignment.

5.2 Extra(s)

The experiments: roundrobin.slip & roundrobin-bug.slip are one additional type of experiment, expanding the concept of the producer-consumer scenario. The value of the name variable passed to the ProduceItem method showcases which producer produced the item.

The circle.slip experiments, consists of three producers, two consumers and one organizer process. The organizer, will depending on the state of the buffer, produce or consume the item in said buffer. The coroutine to which responsibility is given, depends on the prod-ctr and cons-ctr values.