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.

3 First Iteration

This subsection will discuss the 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, used 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 is 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 are taken. The body is evaluated using the evaluate_inline_sequence procedure.

3.3 Problems

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

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

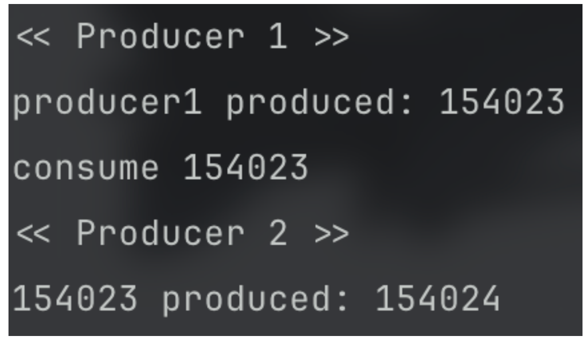

Figure 1: Round robin Environment bug

A similar issues occurs when executing the pingpong2.slip experiment, the output is continuously ping instead of the expected ping, pong … output.

4 Implementation

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 separated.

4.2 Compilation

During the compilation step is 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 only of the name and process_stack value. 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;

  coroutine = make_COR(Name, Context, Main_Empty_Vector, Main_Empty_Vector);

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

Listing 2: Evaluation - Make coroutine

The env and frm values of the COR_type are set as the empty vector during initial evaluation moment.

4.4 Transfer

The expected runtime values are now of the COR_type, once the grammar tags are confirmed for both values, the execution may continue. The context check, instead of checking on the env field, now checks on the cnt field.

To initiate the coroutines, runtime values of the same coroutine are passed to the transfer function. The only action that is needed to take place is calling the Thread_Replace with the to_context as the value. This will call the continue_newprocess_body procedure and execute the body of the coroutine.

In case the runtime values are not the same, the current point of continuation is saved by calling Thread_Keep. The from_process is updated with the new cnt value. The current environment & frame are saved by calling in order: Environment_Get_Environment and Environment_Get_Frame.

From the to_process, the env & frm values are retrieved, if they are not empty (not first iteration), the environment & frame are set to saved values. The program continues with again calling Thread_Replace with to_context as the value.

4.5 Problems

The current implementation, for the ping-pong2.slip experiment outputs a continues stream of ping values, but in the beginning of the process performs the start and one context switch as expected. This situation can be seen in image Figure 2.

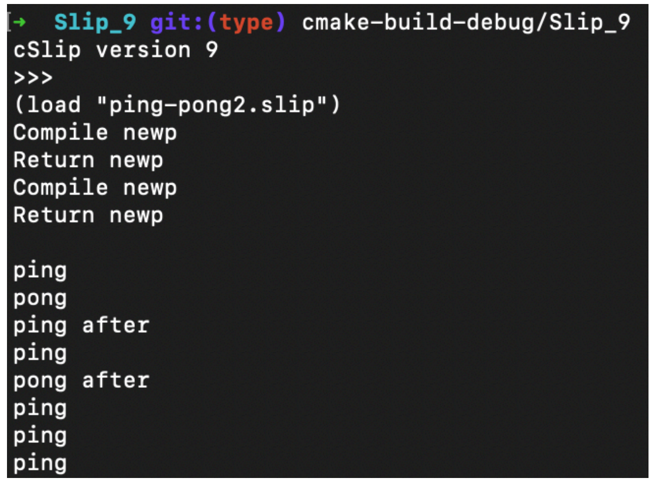

Figure 2: Ping Pong 2 - Bug

The bug alluded to earlier in Figure 1 is still present in this implementation. But the process of swapping environments & frames now takes place in contrast to previous implementation.

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

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

Newprocesses can only be set by using the set! expression.

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 is one additional experiment, expanding the concept of the producer-consumer.

The value of the name variable passed to the ProduceItem, in the second producer, receives the value of the item just produced & consumed by the previous coroutines.