Proof Optimizations

Kristina Sojakova Mihai Codescu

Acknowledgement

This project was funded through the NGI0 Core Fund, a fund established by NLnet with financial support from the European Commission's Next Generation Internet programme, under the aegis of DG Communications Networks, Content and Technology under grant agreement No. 101092990.

1 Introduction

The main aim of this milestone was to improve the performance of the IPDL tool. While it outperforms the old Coq implementation, certain issues have arised, especially on large case studies like the multi-party GMW-N. Before any optimization, the IPDL proof for this case study was obtained in a little over 12 minutes.

2 Parsing

The first problem we have noticed is that the Maude parser is inefficient on large inputs. We have concluded several experiments with Maude parsing, leading to the following results.

A simple but effective solution was to make the parsing lazy, in the sense that the input text is scanned until the declaration terminator "." is encountered.

Since the grammar for IPDL proofs is more complex, with more constructors, a similar solution did not provide the same results for proofs, after changing the syntax such that proofs are also ended with a dot. We have however noticed, when extending the language with multi-file proofs, that we obtain a performance increase when writing

```
proofStep1 then
proofStep2
```

as

```
proofStep1

proofStep2
```

The semantics of the two constructions is the same, because IPDL handles the second proof as "first execute proofStep1, then execute proofStep2 from the configuration reached after executing proofStep2", which coincides with the semantics of `then`.

We can only do this at top level, if a proof step takes a proof as argument, we cannot replace `then` with newline in the argument proof.

Finally, another runtime improvement can be obtained by allowing the user to control whether declared protocols typecheck, which is done via traversing the parse tree of the protocol (because we want to issue meaningful messages pointing to where the errors occur). This can be time consuming for large protocols. The issue of checking whether a protocol is correct is orthogonal to the proof, and we can separate the two.

We have done this by adding a flag `enable-typechecking`. The methodology is to have type checking enabled until the protocol is fully written and verified as correct, and then to remove the flag from the source file.

The results of applying these optimizations to the GMW-N case study are, in chronological order:

- disabling type checking lowered the runtime to ca. 10 minutes;
- removing `then` from the top-level of the proof reduced the runtime to a little more than 9 minutes;
- making parsing lazy further reduced the runtime to 6 and half minutes.

We plan further investigations on improving the data structures used for storing the parse tree of a protocol, jointly with the author of SpeX.

3 Lemmas

We have introduced the following syntax for lemma declarations:

```
lemma NAME = P1 => P2 : PROOF .
```

where `P1, P2` are protocols (or protocol names) and `PROOF` is an IPDL proof that P1 rewrites to P2.

When encountering a lemma declaration, the tool introduces the following in the Maude module stored in the environment:

- a Maude strategy for applying the proof, in the same way as for IPDL subproofs;
- a Maude rewrite rule that directly rewrites `P1` to `P2`
- a strategy for applying this rewrite rule under the IPDL congruence rules for new declarations and parallel composition.

This allows us to handle two execution modes for lemmas, one that runs its proof and the other that check that the proof has already been executed and does not repeat the proof if this is the case. We have implemented this by further extending the Maude module stored in the environment with:

- a strategy with the same name as the lemma, performing no action (i.e., it is defined as `idle`), to be used as a flag and which is added to the module as soon as we called the strategy that applies the proof of the lemma;
- a top-level strategy for executing the lemma, that checks whether the strategy introduced at the previous step is present in the module. If this is the case, we can call the strategy that directly rewrites `P1` to `P2`. If this is not the case, we call the strategy that applies the proof of the lemma, and this also adds the flag strategy, for further applications of the lemma.

4 Multi-file Proofs

For convenience, we can split IPDL among multiple files. We have introduced at the level of SpeX an `include` command that parses the content of a file and

We can use `include` after any input that has been completely processed. For example, this means that we cannot write

```
sym from P over delta (include F)
```

In such a case, we can do the entire symmetry proof inside F and replace the line above with `include F`. Similar restrictions apply to all other IPDL proof steps with subproofs.

The specification methodology will be as folllows. For each fragment `F` we will have

- a proof file `F.proof` that contains the only the IPDL proof to be performed, as a sequence of proof steps;
- a local test file `F.ipdl` that allows us to execute the proof fragment, of the general shape

  ```
  lang IPDL
  include F-definitions.ipdl
  include F-subproofs.ipdl

  start with P over delta

  include F.proof

  check-proof Q
  ```

where the files `definitions.ipdl` and `subproofs.ipdl` include all the definitions and subproofs and lemmas needed in the proof, respectively. In practice,

these files will often be modular themselves, as we may want to include e.g. the declaration of a channel context multiple times.

The main proof file will be of the general shape

```
lang IPDL
include definitions.ipdl
include subproofs.ipdl

start with real over delta

include F1.proof

...

include FN.proof

check-proof idealPlusSim
```

where the files `definitions.ipdl` and `subproofs.ipdl` include the fragments of the files `F-definitions.ipdl` and `F-subproofs.ipdl` that are needed for the main proof. We may want to omit e.g. declarations of protocols used in intermediate checks, for performance issues.