A Formal Proof Generator from Semi-formal Proof Documents

Adrián Riesco[1] Kazuhiro Ogata[2,3]

Facultad de Informática, Universidad Complutense de Madrid, Spain
ariesco@fdi.ucm.es

School of Information Science, JAIST, Japan

Research Center for Software Verification, JAIST, Japan
ogata@jaist.ac.jp

ICTAC 2017
Hanoi, Vietnam

Motivation: CafeOBJ

- CafeOBJ is a language for writing formal specifications and verifying properties of them.
- It implements equational logic by rewriting.
- CafeOBJ specifications are executable, so the specifier can analyze how different terms are reduced.
- In particular, specifiers can write proof scores to prove properties on their specifications.

Motivation: Proof scores

- Proof scores are proof outlines written in CafeOBJ.
- If all proof scores return the expected value when executed (usually `true`), then the corresponding theorems are proved.
- This approach is known as "proving as programming."

Motivation: Proof scores

- An important advantage of this approach is its flexibility: the syntax for performing proofs is the same as for specifying systems.
- However, we lose formality because CafeOBJ does not check proof scores in any way.
- For this reason, in this paper we present:
  - An inductive theorem prover.
  - A proof script generator that infers formal proofs from proof scores.
- These tools extend the CafeInMaude compiler, implemented in Maude.
- CafeInMaude takes advantage of Maude metalevel and stores a metarepresentation of proof scores, so we can reason with them at the metalevel.

Proving with proof scores

- Let's see how to verify part of a simple mutual exclusion protocol for two processes.
- We define the labels assigned to each process: that are `rs` and `cs` (*remainder section* and *critical section*).

```
mod! LABEL {
  [Label]
  ops rs cs : -> Label {constr}
  eq (rs = cs) = false .
}
```

Proving with proof scores

- We define the constructors for the system (`Sys`).
- `init` stands for the initial state.
- `enter1` and `enter2` indicate that the first process and the second process want to enter the critical section, respectively.
- `leave1` and `leave2` indicate that they want to leave the critical section.

```
mod* 2P-MUTEX {
  pr(LABEL)
  [Sys]
-- any initial state
  op init : -> Sys {constr}
-- transitions
  ops enter1 enter2 : Sys -> Sys {constr}
  ops leave1 leave2 : Sys -> Sys {constr}
```

Proving with proof scores

- Observations on the system are obtained by using `pc1` for the first process and `pc2` for the second one:

```
ops pc1 pc2 : Sys -> Label
```

- The observations for the `init` state are both `rs`:

```
eq pc1(init) = rs .
eq pc2(init) = rs .
```

- In the following we focus on the behavior when entering the critical section.

Proving with proof scores

- We define `c-enter1` and `c-enter2` to check whether a process can enter the critical section:

```
ops c-enter1 c-enter2 : Sys -> Bool
eq c-enter1(S) = (pc2(S) = rs) .
eq c-enter2(S) = (pc1(S) = rs) .
```

- We use `c-enter1` to define the behavior for `enter1`:

```
ceq pc1(enter1(S)) = cs   if c-enter1(S) .
ceq pc2(enter1(S)) = rs   if c-enter1(S) .
ceq enter1(S)      = S    if not c-enter1(S) .
```

- The observations for `enter2` are defined in the same way using `c-enter2`.

Proving with proof scores

- We define an invariant stating that both processes cannot be at the critical section at the same time:

```
op inv1 : Sys -> Bool
eq inv1(S) = not ((pc1(S) = cs) and (pc2(S) = cs)) .
```

- How to prove this property?

Proving with proof scores

- We first prove the invariant for the `init` state:

```
open 2P-MUTEX .
  red inv1(init) .
close

Result: true
```

- However, a similar proof score for `inv1(enter1(s))` would fail:

```
open 2P-MUTEX .

  op s : -> Sys .
  red inv1(enter1(s)) .
close

Result: true xor(cs = pc1(enter1(s))) and cs = pc2(enter1(s)): Bool
```

Proving with proof scores

- We need to enrich the proof score with a case splitting and an implication:

```
open 2P-MUTEX .
  op s : -> Sys .
  eq pc2(s) = rs .

  red inv1(s) implies inv1(enter1(s)) .
close
```

- What if
  - We forget the complementary case (`(pc2(s) = rs) = false`)?
  - We forget inductive cases (e.g. `enter2`)?
  - We use the implication with any other function?
- CafeOBJ does not check any of the above, so it is easy to miss a case.

Interactive theorem proving

- The standard solution to this lack of formality is using an interactive theorem prover.
- We have developed the CafeInMaudeProofAssistant (*CiMPA*). It supports:
  - Several equations as goals.
  - Induction on constructors.
  - Theorem of constants.
  - Case splitting by `true`/false.
  - Case splitting by constructors.
  - Implication with induction hypotheses.
  - Discharge goals by applying reduction.
- CiMPA is implemented using Maude metalevel.
- Each node of the proof tree contains the module where the equations for the hypotheses and the case splitting thus far have been added.

Interactive theorem proving

- Let's see how to prove part of our protocol with CiMPA.
- Goals are introduced by using the `:goal` command:

```
open 2P-MUTEX .
  :goal{
   eq [inv1 :nonexec] : inv1(S:Sys) = true .
  }
```

- We can apply induction on variables as:

```
:ind on (S:Sys)
:apply(si)
```

- This command generates 5 new goals.
- They follow their alphabetic ordering, so CiMPA starts by the subgoal for `enter1(S#Sys)`.

Interactive theorem proving

- We ask now CiMPA to apply case splitting distinguishing whether the equation `eq pc2(S#Sys) = rs` holds:

```
:def csb1 = :ctf {eq pc2(S#Sys) = rs .}
:apply(csb1)
```

- Now we use an implication with the induction hypothesis as premise:

```
:imp [inv1] .
```

- We have reached the leaf in the proof tree that corresponds to the proof score for `enter1`.
- We can discharge the goal by using equations:

```
:apply (rd)
```

CiMPG

- We need to choose among flexibility in proof scores or formality in CiMPA.
- Proof scores are more flexible, so it is easy to work with them.
- They are only validated by a *careful examination*, so there are no guarantees and it is more difficult to convince others.
- We need a translation from proof scores to CiMPA proof scripts.
- We have implemented this functionality in the CafeInMaudeProofGenerator (CiMPG).

CiMPG

- CiMPG provides annotations for identifying proof scores proof scripts for CiMPA from them.
- The restrictions imposed to these proof scores are:
  - Reducing only goal-related terms.
  - It ensures that no dummy goals are generated.
  - Making sure that all environments import the same module.
  - It is required to make sure that the property is being proved for the same specification.
  - Relying in functionalities that can be simulated by CiMPA commands.
  - It ensures that the proof script can be generated.

CiMPG

- These annotations are of the form `:id(LAB)`:

```
open 2P-MUTEX .
 :id(2p-mutex)  --- CiMPG annotation

 op s : -> Sys .
 eq pc2(s) = rs .

 red inv(s)implies inv(enter1(s)) .
close
```

- The annotation `:proof(LAB)` is used to generate the script:

```
open 2P-MUTEX
 :proof(2p-mutex)
close
```

CiMPG

- CiMPG follows a metalevel algorithm to infer the proof.
- It first infers the goal to be proved by generalizing the reduction commands in the proof scores.
- Then, it starts a loop that will modify the tree by
  1. Looking for those proof scores related to the current goal.
  2. Enriching the module.
- It finds the appropriate proof scores by
  1. Checking the goal in the proof score distinguish the same case as the current node of the proof tree.
  2. The splittings do not refer to different cases.

CiMPG

- The module is enriched by:
  1. Using induction.
  2. Applying the theorem of constants.
  3. Performing case splitting.
  4. Modifying the goal (by using implications).
- Once the module and the goal are equal to the ones in the proof score, the goal can be discharged by using equations.
- Note that the order for asserting cases is important, since an erroneous one will generate modules do not correspond to those in the proof scores.
- The loop finishes when no more proof scores can be used.

CiMPG

- How does CiMPG help specifiers?
- If we try to obtain the proof script from our original proof scores (where a case for `enter1`) was missing CiMPG generates a `postpone` command.
- If we ask CiMPA to show the state of the proof at that point it displays:

```
Next goal is 1-2: eq [inv :nonexec]: inv(enter1(S#Sys)) = true .
  -- Assumptions:
  eq pc2(S#Sys)= rs  = false .
  eq [inv :nonexec]: inv(S#Sys) = true .
```

Benchmarks

- The benchmarks performed thus far give us confidence in its applicability.

Name	LOC	Commands	Time	Comments
2p-mutex	50 + 70	28	470 ms	Mutual exclusion protocol
TAS	50 + 230	76	2130 ms	Spinlock
QLOCK	100 + 400	112	9510 ms	Dijkstra's binary semaphore
NSLPK	180 + 1100	284	48390 ms	Authentication protocol NSLPK
Cloud	120 + 1700	383	96470 ms	Cloud synchronization protocol

Concluding remarks

- We have presented two tools that combine different approaches to theorem proving in CafeOBJ.
- This combination is sound and, even though it is not complete, the examples used thus far give us confidence in the technique.
- In contrast to other approaches, which translate logics to take advantage of different provers, CiMPG translates proofs.

Ongoing work

- We are working in more commands for performing different kinds of case splitting when dealing with sequences.
- Once they are proved sound and added to CiMPA we plan to include them into the CiMPG inference algorithm.
- It would be also interesting to consider proof scores involving searches or unification.
- These extensions will allow us to analyze a new set of CafeOBJ proofs.