Some Tips on Writing Proof Scores in the OTS/CafeOBJ Method

Kazuhiro Ogata[1,2] and Kokichi Futatsugi[2]

[1] NEC Software Hokuriku, Ltd.
ogatak@acm.org
[2] Japan Advanced Institute of Science and Technology (JAIST)
kokichi@jaist.ac.jp

Abstract. The OTS/CafeOBJ method is an instance of the proof score approach to systems analysis, which has been mainly devoted by researchers in the OBJ community. We describe some tips on writing proof scores in the OTS/CafeOBJ method and use a mutual exclusion protocol to exemplify the tips. We also argue soundness of proof scores in the OTS/CafeOBJ method.

1 Introduction

The proof score approach to systems analysis has been mainly devoted by researchers in the OBJ community [10, 8]. In the approach, an executable algebraic specification language is used to specify systems and system properties, and a processor of the language, which has a rewrite engine as one of its functionalities, is used as a proof assistant to prove that systems satisfy system properties. Proof plans called *proof scores* are written in the algebraic specification language to conduct such proofs and the proof scores are executed by the language processor by means of rewriting to check if the proofs are success.

Proof scores can be regarded as programs to prove that algebraic specifications satisfy system properties. While proof scores are being designed, constructed and debugged, we can understand algebraic specifications being analyzed more profoundly, which may even let us find flaws lurked in the specifications [15, 14]. Our thought on proof is similar to that of the designers of LP [11]. Proof scripts written in a tactic language provided by proof assistants such as Coq [1] and Isabel/HOL [13] may be regarded as such programs, but it seems that such proof assistants rather aim for mechanizing mathematics.

We have argued that the proof score approach to systems analysis is an attractive approach to design verification in [6] thanks to (1) balanced human-computer interaction and (2) flexible but clear structure of proof scores. The former means that humans are able to focus on proof plans, while tedious and detailed computations can be left to computers; humans do not necessarily have to know what deductive rules or equations should be applied to goals to prove. The latter means that lemmas do not need to be proved in advance and proof scores can help humans comprehend the corresponding proofs; a proof that a

system satisfies a system property can be conducted even when all lemmas used have not been proved, and assumptions used are explicitly and clearly written in proof scores. To precisely assess the achievement of (1) and (2) in the proof score approach and compare it with systems analysis with other existing proof assistants, however, we need further studies.

The OTS/CafeOBJ method [17, 4, 7] is an instance of the proof score approach to systems analysis. In the OTS/CafeOBJ method, observational transition systems (OTSs) are used as models of systems and CafeOBJ [2], an executable algebraic specification language/system, is used; OTSs are transition systems, which are straightforwardly written as algebraic specifications. An older version of the OTS/CafeOBJ method is described in [17, 4], and the latest version is described in [7]. We have conducted case studies, among which are [15, 18, 19, 16, 20], to demonstrate the usefulness of the OTS/CafeOBJ method. In this paper, we describe some tips on writing proof scores in the OTS/CafeOBJ method. A mutual exclusion protocol called *Tlock* using *atomicInc*, which atomically increments the number stored in a variable and returns the old number, is used as an example. We also argue soundness of proof scores in the OTS/CafeOBJ method.

The rest of the paper is organized as follows. Section 2 describes the OTS/CafeOBJ method. Section 3 describes tips on writing proof scores in the OTS/CafeOBJ method. Section 4 informally argue soundness of proof scores in the OTS/CafeOBJ method. Section 5 concludes the paper.

2 The OTS/CafeOBJ Method

In the OTS/CafeOBJ method, systems are analyzed as follows.

1. Model a system as an OTS $\mathcal{S}$.
2. Write $\mathcal{S}$ in CafeOBJ as an algebraic specification. The specification consists of sorts (or types), operators on the sorts, and equations that define (properties of) the operators. The specification can be executed by using equations as left-to-right rewrite rules by CafeOBJ.
3. Write system properties in CafeOBJ. Let P be the set of such system properties and let P' be the empty set. .
4. If P is empty, the analysis has been successfully finished, which means that $\mathcal{S}$ satisfies all the properties in P'. Otherwise, extract a property p from P and go next. .
5. Write a proof score in CafeOBJ to prove that $\mathcal{S}$ satisfies p. The proof may need other system properties as lemmas. Write such system properties in CafeOBJ and put them that are not in P' into P if any. .
6. Execute (or play) the proof score with CafeOBJ. If all the results are as expected, then the proof is discharged. Put p into P' and go to 4. If all the results are not as expected, rewrite the proof score and repeat 6. .

Tasks 5 and 6 may be interactively conducted together. A counterexample may be found in tasks 5 and 6.

In this section, we mention CafeOBJ, describe the definitions of basic concepts on OTSs, write on how to write OTSs in CafeOBJ and how to write proof scores that OTSs satisfy invariant properties in CafeOBJ.

2.1 CafeOBJ

CafeOBJ [2] is an algebraic specification language/system mainly based on order-sorted algebras and hidden algebras [9, 3]. Abstract data types are specified in terms of order-sorted algebras, and abstract machines are specified in terms of hidden algebras. Algebraic specifications of abstract machines are called *behavioral specifications*. There are two kinds of sorts in CafeOBJ: *visible sorts* and *hidden sorts*. A visible sort denotes an abstract data type, while a hidden sort denotes the state space of an abstract machine. There are three kinds of operators (or operations) with respect to (wrt) hidden sorts: *hidden constants*, *action operators* and *observation operators*. Hidden constants denote initial states of abstract machines, action operators denote state transitions of abstract machines, and observation operators let us know the situation where abstract machines are located. Both an action operator and an observation operator take a state of an abstract machine and zero or more data. The action operator returns the successor state of the state wrt the state transition denoted by the action operator plus the data. The observation operator returns a value that characterizes the situation where the abstract machine is located.

Basic units of CafeOBJ specifications are modules. CafeOBJ provides built-in modules. One of the most important built-in modules is `BOOL` in which propositional logic is specified. `BOOL` is automatically imported by almost every module unless otherwise stated. In `BOOL` and its parent modules, declared are the visible sort `Bool`, the constants `true` and `false` of `Bool`, and operators denoting some basic logical connectives. Among the operators are `not_`, `_and_`, `_or_`, `_xor_`, `_implies_` and `_iff_` denoting negation ($\neg$), conjunction ($\wedge$), disjunction ($\vee$), exclusive disjunction (xor), implication ($\Rightarrow$) and logical equivalence ($\Leftrightarrow$), respectively. The operator `if_then_else_fi` corresponding to the **if** construct in programming languages is also declared. CafeOBJ uses the Hsiang term rewriting system (TRS) [12] as the decision procedure for propositional logic, which is implemented in `BOOL`. CafeOBJ reduces any term denoting a proposition that is always true (false) to `true` (`false`). More generally, a term denoting a proposition reduces to an exclusively disjunctive normal form of the proposition.

2.2 Observational Transition Systems (OTSs)

We suppose that there exists a universal state space denoted Υ and that each data type used in OTSs is provided. The data types include Bool for truth values. A data type is denoted D_*.

Definition 1 (OTSs). *An OTS $\mathcal{S}$ is $\langle \mathcal{O}, \mathcal{I}, \mathcal{T} \rangle$ such that*

- *$\mathcal{O}$: A finite set of observers. Each* observer *$o_{x_1:D_{o1},\ldots,x_m:D_{om}} : \Upsilon \rightarrow D_o$ is an indexed function that has m indexes $x_1, \ldots, x_m$ whose types are*

$D_{o1}, \ldots, D_{om}$. *The equivalence relation* ($\upsilon_1 =_{\mathcal{S}} \upsilon_2$) *between two states* $\upsilon_1, \upsilon_2 \in \Upsilon$ *is defined as* $\forall o_{x_1,\ldots,x_m} : \mathcal{O}.\, (o_{x_1,\ldots,x_m}(\upsilon_1) = o_{x_1,\ldots,x_m}(\upsilon_2))$, *where* $\forall o_{x_1,\ldots,x_m} : \mathcal{O}$ *is the abbreviation of* $\forall o_{x_1,\ldots,x_m} : \mathcal{O}. \forall x_1 : D_{o1} \ldots \forall x_m : D_{om}$.
- $\mathcal{I}$ *: The set of initial states such that* $\mathcal{I} \subseteq \Upsilon$.
- $\mathcal{T}$ *: A finite set of transitions. Each* transition $t_{y_1:D_{t1},\ldots,y_n:D_{tn}} : \Upsilon \rightarrow \Upsilon$ *is an indexed function that has* n *indexes* $y_1, \ldots, y_n$ *whose types are* $D_{t1}, \ldots, D_{tn}$ *provided that* $t_{y_1,\ldots,y_n}(\upsilon_1) =_{\mathcal{S}} t_{y_1,\ldots,y_n}(\upsilon_2)$ *for each* $[\upsilon] \in \Upsilon/=_{\mathcal{S}}$, *each* $\upsilon_1, \upsilon_2 \in [\upsilon]$ *and each* $y_k : D_{tk}$ *for* $k = 1, \ldots, n$. $t_{y_1,\ldots,y_n}(\upsilon)$ *is called* the successor state *of* υ *wrt* $\mathcal{S}$. *Each transition* $t_{y_1,\ldots,y_n}$ *has the condition* $c\text{-}t_{y_1:D_{t1},\ldots,y_n:D_{tn}} : \Upsilon \rightarrow$ Bool, *which is called* the effective condition *of the transition. If* $c\text{-}t_{y_1,\ldots,y_n}(\upsilon)$ *does not hold, then* $t_{y_1,\ldots,y_n}(\upsilon) =_{\mathcal{S}} \upsilon$. □

We note the following two points on transitions, which have something to do with writing proof scores. (1) Although transitions are defined as relations among states in some other existing transition systems, transitions are functions on states in OTSs. This is because transitions are represented by (action) operators in behavioral specifications and operators are functions in CafeOBJ. However, multiple transitions that are functions on states can be substituted for one transition that is a relation among states. (2) Basically there is no restriction on the form of effective conditions. But, effective conditions should be in the form $c_1\text{-}t_{y_1,\ldots,y_n}(\upsilon) \wedge \ldots \wedge c_M\text{-}t_{y_1,\ldots,y_n}(\upsilon)$, where each $c_k\text{-}t_{y_1,\ldots,y_n}(\upsilon)$ has no logical connectives or has one negation at head, so that proof scores can have clear structure. When an effective condition is not in this form, it is converted to a disjunctive normal form. If the disjunctive normal form has more than one disjunct, multiple transitions each of which has one of the disjuncts as its effective condition can be substituted for the corresponding transition.

Definition 2 (Reachable states). *Given an OTS* $\mathcal{S}$, reachable states *wrt* $\mathcal{S}$ *are inductively defined:*

- *Each* $\upsilon_{\text{init}} \in \mathcal{I}$ *is reachable wrt* $\mathcal{S}$.
- *For each* $t_{y_1,\ldots,y_n} \in \mathcal{T}$ *and each* $y_k : D_{tk}$ *for* $k = 1. \ldots, n$, $t_{x_1,\ldots,x_n}(\upsilon)$ *is reachable wrt* $\mathcal{S}$ *if* $\upsilon \in \Upsilon$ *is reachable wrt* $\mathcal{S}$.

Let $\mathcal{R}_{\mathcal{S}}$ *be the set of all reachable states wrt* $\mathcal{S}$. □

Predicates whose types are $\Upsilon \rightarrow$ Bool are called *state predicates*. All properties considered in this paper are *invariants*.

Definition 3 (Invariants). *Any state predicate* $p : \Upsilon \rightarrow$ *Bool is called* invariant *wrt* $\mathcal{S}$ *if* p *holds in all reachable states wrt* $\mathcal{S}$, *i.e.* $\forall \upsilon : \mathcal{R}_{\mathcal{S}}.\, p(\upsilon)$. □

We suppose that each state predicate p considered in this paper has the form $\forall z_1 : D_{p1} \ldots \forall z_a : D_{pa}.\, P(\upsilon, z_1, \ldots, z_a)$, where $\upsilon, z_1, \ldots, z_a$ are all variables in p and $P(\upsilon, z_1, \ldots, z_a)$ does not contain any quantifiers.

A concrete example of how to model a system as an OTS is given.

Example 1 (Tlock). The pseudo-code executed by each process i can be written as follows:

Loop
l1: $ticket[i] := atomicInc(tvm)$;
l2: **repeat until** $ticket[i] = turn$;
Critical section;
cs: $turn := turn + 1$;

tvm and *turn* are non-negative integer variables shared by all processes and $ticket[i]$ is a non-negative integer variable that is local to process i. Initially, each process i is at label l1, *tvm* and *turn* are 0, and $ticket[i]$ for each i is unspecified. The value of *tvm* (which stands for a ticket vending machine) is the next available ticket. Each process i obtains a ticket, which is stored in $ticket[i]$, at label l1. A process is allowed to enter the critical section if its ticket equals the value of *turn* at label l2. *turn* is incremented when a process leaves the critical section at label cs.

Let Label, Pid and Nat be the types of labels (l1, l2 and cs), process IDs and non-negative integers (natural numbers). Tlock can be modeled as the OTS $\mathcal{S}_{\mathrm{Tlock}}$ such that

- $\mathcal{O}_{\mathrm{Tlock}} \triangleq \{\mathrm{tvm} : \Upsilon \to \mathrm{Nat}, \mathrm{turn} : \Upsilon \to \mathrm{Nat}, \mathrm{ticket}_{i:\mathrm{Pid}} : \Upsilon \to \mathrm{Nat}, \mathrm{pc}_{i:\mathrm{Pid}} : \Upsilon \to \mathrm{Label}\}$
- $\mathcal{I}_{\mathrm{Tlock}} \triangleq \{o_{\mathrm{init}} \in \Upsilon \mid \mathrm{tvm}(o_{\mathrm{init}}) = 0 \wedge \mathrm{turn}(o_{\mathrm{init}}) = 0 \wedge \forall i : \mathrm{Pid}.\, (\mathrm{pc}_i(o_{\mathrm{init}}) = \mathrm{l1})\}$
- $\mathcal{T}_{\mathrm{Tlock}} \triangleq \{\mathrm{get}_{i:\mathrm{Pid}} : \Upsilon \to \Upsilon, \mathrm{check}_{i:\mathrm{Pid}} : \Upsilon \to \Upsilon, \mathrm{exit}_{i:\mathrm{Pid}} : \Upsilon \to \Upsilon\}$

The three transitions are defined as follows:

- get_i : $\text{c-get}_i(o) \triangleq \mathrm{pc}_i(o) = \mathrm{l1}$. If $\text{c-want}_i(o)$, then

  $\mathrm{tvm}(\mathrm{get}_i(o)) \triangleq \mathrm{tvm}(o) + 1$, $\mathrm{turn}(\mathrm{get}_i(o)) \triangleq \mathrm{turn}(o)$,
  $\mathrm{ticket}_j(\mathrm{get}_i(o)) \triangleq$ **if** $i = j$ $\mathrm{tvm}(o)$ **else** $\mathrm{ticket}_j(o)$, and
  $\mathrm{pc}_j(\mathrm{get}_i(o)) \triangleq$ **if** $i = j$ **then** l2 **else** $\mathrm{pc}_j(o)$.

- check_i : $\text{c-check}_i(o) \triangleq \mathrm{pc}_i(o) = \mathrm{l2} \wedge \mathrm{ticket}_i(o) = \mathrm{turn}(o)$. If $\text{c-want}_i(o)$, then

  $\mathrm{tvm}(\mathrm{check}_i(o)) \triangleq \mathrm{tvm}(o)$, $\mathrm{turn}(\mathrm{check}_i(o)) \triangleq \mathrm{turn}(o)$,
  $\mathrm{ticket}_j(\mathrm{get}_i(o)) \triangleq \mathrm{ticket}_j(o)$, and $\mathrm{pc}_j(\mathrm{check}_i(o)) \triangleq$ **if** $i = j$ **then** cs **else** $\mathrm{pc}_j(o)$.

- exit_i : $\text{c-exit}_i(o) \triangleq \mathrm{pc}_i(o) = \mathrm{cs}$. If $\text{c-want}_i(o)$, then

  $\mathrm{tvm}(\mathrm{exit}_i(o)) \triangleq \mathrm{tvm}(o) + 1$, $\mathrm{turn}(\mathrm{exit}_i(o)) \triangleq \mathrm{turn}(o)$,
  $\mathrm{ticket}_j(\mathrm{get}_i(o)) \triangleq \mathrm{ticket}_j(o)$, and $\mathrm{pc}_j(\mathrm{exit}_i(o)) \triangleq$ **if** $i = j$ **then** l1 **else** $\mathrm{pc}_j(o)$.

Let $\mathrm{MX}(o)$ be $\forall i, j : \mathrm{Pid}.\, [(pc_i(o) = cs \wedge pc_j(o) = cs) \Rightarrow i = j]$. $\mathrm{MX}(o)$ is invariant wrt $\mathcal{S}_{\mathrm{Tlock}}$, i.e. $\forall o : \mathcal{R}_{\mathcal{S}_{\mathrm{Tlock}}}.\, \mathrm{MX}(o)$, although it may need to be verified. □

2.3 Specifying OTSs in CafeOBJ

We suppose that a visible sort $\mathtt{V}_*$ corresponding to each data type D_* used in OTSs and the related operators are provided. $\mathtt{X}_k$ and $\mathtt{Y}_k$ are CafeOBJ variables corresponding to indexes x_k and y_k of observers and transitions, respectively.

The universal state space Υ is represented by a hidden sort, say `H` declared as `*[H]*` by enclosing it with `*[` and `]*`. Given an OTS $\mathcal{S}$, an arbitrary initial state is represented by a hidden constant, say `init`, each observer $o_{x_1,\ldots,x_m}$ is represented by an observation operator, say `o`, and each transition $t_{y_1,\ldots,y_n}$ is represented by an action operator, say `t`. The hidden constant `init`, the observation operator `o` and the action operator `t` are declared as follows:

```
op init : -> H
bop o : H V_o1 ...V_om -> V_o
bop t : H V_t1 ...V_tn -> H
```

The keyword `bop` or `bops` is used to declare observation and action operators.

We suppose that the value returned by $o_{x_1,\ldots,x_m}$ in an arbitrary initial state can be expressed as $f(x_1,\ldots,x_m)$. This is expressed by the following equation:

```
eq o(init,X_1,...,X_m) = f(X_1,...,X_m) .
```

`f(`X_1`,...,`X_m`)` is the CafeOBJ term corresponding to $f(x_1,\ldots,x_m)$.

Each transition $t_{y_1,\ldots,y_n}$ is defined by describing what the value returned by each observer $o_{x_1,\ldots,x_m}$ in the successor state becomes when $t_{y_1,\ldots,y_n}$ is applied in a state υ. When $c\text{-}t_{y_1,\ldots,y_n}(\upsilon)$ holds, this is expressed generally by a conditional equation that has the form

```
ceq o(t(S,Y_1,...,Y_n),X_1,...,X_m) = e-t(S,Y_1,...,Y_n,X_1,...,X_m)
    if c-t(S,Y_1,...,Y_n) .
```

`S` is a CafeOBJ variable of `H`, corresponding to υ. `e-t(S,`Y_1`,...,`Y_n`,`X_1`,...,`X_m`)` is the CafeOBJ term corresponding to the value returned by $o_{x_1,\ldots,x_m}$ in the successor state denoted by `t(S,`Y_1`,...,`Y_n`)`. `c-t(S,`Y_1`,...,`Y_n`)` is the CafeOBJ term corresponding to $c\text{-}t_{y_1,\ldots,y_n}(\upsilon)$.

If $c\text{-}t_{y_1,\ldots,y_n}(\upsilon)$ always holds in any state υ or the value returned by $o_{x_1,\ldots,x_m}$ is not affected by applying $t_{y_1,\ldots,y_n}$ in any state υ (i.e. regardless of the truth value of $c\text{-}t_{y_1,\ldots,y_n}(\upsilon)$), then a usual equation is used instead of a conditional equation. The usual equation has the form

```
eq o(t(S,Y_1,...,Y_n),X_1,...,X_m) = e-t(S,Y_1,...,Y_n,X_1,...,X_m) .
```

`e-t(S,`Y_1`,...,`Y_n`,`X_1`,...,`X_m`)` is `S` if the value returned by $o_{x_1,\ldots,x_m}$ is not affected by applying $t_{y_1,\ldots,y_n}$ in any state.

When $c\text{-}t_{y_1,\ldots,y_n}(\upsilon)$ does not hold, $t_{y_1,\ldots,y_n}$ changes nothing, which is expressed by a conditional equation that has the form

```
ceq t(S,Y_1,...,Y_n) = S if not c-t(S,Y_1,...,Y_n) .
```

We give the CafeOBJ specification of $\mathcal{S}_{\mathrm{Tlock}}$.

Example 2 (CafeOBJ specification of $\mathcal{S}_{\mathrm{Tlock}}$). $\mathcal{S}_{\mathrm{Qlock}}$ is specified in CafeOBJ as the module `TLOCK`:

```
mod* TLOCK {  pr(PNAT) pr(LABEL) pr(PID)
  *[Sys]*
-- an arbitrary initial state
  op init : -> Sys
-- observation operators
  bops tvm turn : Sys -> Nat  bop ticket : Sys Pid -> Nat
  bop  pc : Sys Pid -> Label
-- action operators
  bops get check exit : Sys Pid -> Sys
-- CafeOBJ variables
  var S : Sys  vars I J : Pid
-- init
  eq tvm(init) = 0 .  eq turn(init) = 0 .  eq pc(init,I) = l1 .
-- get
  op c-get : Sys Pid -> Bool {strat: (0 1 2)}
  eq c-get(S,I) = (pc(S,I) = l1) .
  --
  ceq tvm(get(S,I)) = s(tvm(S)) if c-get(S,I) .
  eq  turn(get(S,I)) = turn(S) .
  ceq ticket(get(S,I),J)
      = (if I = J then tvm(S) else ticket(S,J) fi) if c-get(S,I) .
  ceq pc(get(S,I),J) = (if I = J then l2 else pc(S,J) fi) if c-get(S,I) .
  ceq get(S,I) = S if not c-get(S,I) .
-- check
  op c-check : Sys Pid -> Bool {strat: (0 1 2)}
  eq c-check(S,I) = (pc(S,I) = l2 and ticket(S,I) = turn(S)) .
  --
  eq  tvm(check(S,I)) = tvm(S) .  eq  turn(check(S,I)) = turn(S) .
  eq  ticket(check(S,I),J) = ticket(S,J) .
  ceq pc(check(S,I),J)
      = (if I = J then cs else pc(S,J) fi) if c-check(S,I) .
  ceq check(S,I) = S if not c-check(S,I) .
-- exit
  op c-exit : Sys Pid -> Bool {strat: (0 1 2)}
  eq c-exit(S,I) = (pc(S,I) = cs) .
  --
  eq  tvm(exit(S,I)) = tvm(S) .
  ceq turn(exit(S,I)) = s(turn(S)) if c-exit(S,I) .
  eq  ticket(exit(S,I),J) = ticket(S,J) .
  ceq pc(exit(S,I),J)
      = (if I = J then l1 else pc(S,J) fi) if c-exit(S,I) .
  ceq exit(S,I) = S if not c-exit(S,I) .
}
```

A comment starts with `--` and terminates at the end of the line. `PNAT`, `LABEL` and `PID` are the modules in which natural numbers, labels and process IDs are specified. The keyword `pr` is used to imports modules. The operator `s` of `s(tvm(S))` and `s(turn(S))` is the successor function of natural numbers. The keyword `start:` is used to specify local strategies to operators [5]. The local strategy `(0 1 2)` given to `c-get` indicates that when CafeOBJ meets a term

whose top is `c-get` such as `c-get(`s`,`i`)`, CafeOBJ should try to rewrite the whole term such as `c-get(`s`,`i`)`. If CafeOBJ does not find any rules with which the term is rewritten, it evaluates the first and second arguments such as s and i in that order, and tries to rewrite the whole term such as `c-get(`s'`,`i'`)` again, where s' and i' are the results obtained by evaluating s and i. □

2.4 Proof Scores of Invariants

Although some invariants may be proved by rewriting and/or case splitting only, we often need to use induction, especially *simultaneous induction* [7]. We then describe how to verify $\forall o : \mathcal{R}_{\mathcal{S}}.\, p(o)$ by simultaneous induction by writing proof scores in CafeOBJ based on the CafeOBJ specification of $\mathcal{S}$.

It is often impossible to prove $\forall o : \mathcal{R}_{\mathcal{S}}.\, p(o)$ alone. We then suppose that it is possible to prove $\forall o : \mathcal{R}_{\mathcal{S}}.\, p(o)$ together with $N-1$ other state predicates[3], that is, we prove $\forall o : \mathcal{R}_{\mathcal{S}}.\, (p_1(o) \wedge \ldots \wedge p_N(o))$, where p_1 is p. We suppose that each p_k has the form $\forall z_k : D_{pk}.\, P_k(o, z_k)$ for $k = 1, \ldots, N$. Note that the method described here can be used when p_k has more than one universally quantified variable. Let $o^{\mathrm{c}}_{\mathrm{init}}$ be an arbitrary initial state of $\mathcal{S}$, and then for the base case, all we have to do is to prove

$$\forall z_1 : D_{p1}.\, P_1(o^{\mathrm{c}}_{\mathrm{init}}, z_1) \wedge \ldots \wedge \forall z_N : D_{pN}.\, P_N(o^{\mathrm{c}}_{\mathrm{init}}, z_N) \tag{1}$$

For each induction case (i.e. each $t_{y_1,\ldots,y_n} \in \mathcal{T}$), all we have to do is to prove

$$\begin{array}{l} \forall z_1 : D_{p1}.\, P_1(o^{\mathrm{c}}, z_1) \wedge \ldots \wedge \forall z_N : D_{pN}.\, P_N(o^{\mathrm{c}}, z_N) \\ \Rightarrow \forall z_1 : D_{p1}.\, P_1(t_{y^{\mathrm{c}}_1,\ldots,y^{\mathrm{c}}_n}(o^{\mathrm{c}}), z_1) \wedge \ldots \wedge \forall z_N : D_{pN}.\, P_N(t_{y^{\mathrm{c}}_1,\ldots,y^{\mathrm{c}}_n}(o^{\mathrm{c}}), z_N) \end{array} \tag{2}$$

for an arbitrary state o^{c} and an arbitrary value y^{c}_k for $k = 1, \ldots, n$.

To prove (1), we can separately prove each conjunct

$$P_i(o^{\mathrm{c}}_{\mathrm{init}}, z^{\mathrm{c}}_k) \tag{3}$$

where z^{c}_k is an arbitrary value of D_{pk} for $k = 1, \ldots, N$. To prove (2), assuming $\forall z_1 : D_{p1}.\, P_1(o^{\mathrm{c}}, z_1), \ldots, \forall z_N : D_{pN}.\, P_N(o^{\mathrm{c}}, z_N)$, we can separately prove each $P_k(t_{y^{\mathrm{c}}_1,\ldots,y^{\mathrm{c}}_n}(o^{\mathrm{c}}), z^{\mathrm{c}}_k)$, where z^{c}_k is an arbitrary value of D_{pk}, for $k = 1, \ldots, N$. $P_k(o^{\mathrm{c}}, z^{\mathrm{c}}_k)$ is often used as an assumption to prove $P_k(t_{y^{\mathrm{c}}_1,\ldots,y^{\mathrm{c}}_n}(o^{\mathrm{c}}), z^{\mathrm{c}}_k)$. Therefore, the formula to prove has the form

$$(P_\alpha(o^{\mathrm{c}}, d_\alpha) \wedge P_\beta(o^{\mathrm{c}}, d_\beta) \wedge \ldots) \Rightarrow [P_k(o^{\mathrm{c}}, z^{\mathrm{c}}_k) \Rightarrow P_k(t_{y^{\mathrm{c}}_1,\ldots,y^{\mathrm{c}}_n}(o^{\mathrm{c}}), z^{\mathrm{c}}_k)] \tag{4}$$

where $\alpha, \beta, \ldots \in \{1, \ldots, N\}$ and $d_\alpha, d_\beta, \ldots$ are some values of $D_{p\alpha}, D_{p\beta}, \ldots$ for $i = 1, \ldots, N$.

We next describe how to write proof plans of (3) and (4) in CafeOBJ. We first declare the operators denoting $P_1, \ldots, P_N$ and the equations defining the operators. The operators and equations are declared in a module, say `INV` (which imports the module where $\mathcal{S}$ is written), as follows:

[3] Generally, such $N-1$ state predicates should be found while $\forall o : \mathcal{R}_{\mathcal{S}}.\, p(o)$ is being proved.

```
op invk : H Vpk -> Bool
eq invk(S,Zk) = Pi(S,Zk) .
```

for $k = 1, \ldots, N$. $\mathtt{Z}_k$ is a CafeOBJ variable of $\mathtt{V}_{pk}$ and $\mathtt{P}_i\mathtt{(S,Z}_k\mathtt{)}$ is a CafeOBJ term denoting $P_k(\upsilon, z_k)$. In `INV`, we also declare a constant $\mathtt{z}_k^{\mathrm{c}}$ denoting an arbitrary value of $\mathtt{V}_{pk}$ for $i = 1, \ldots, N$. We then declare the operators denoting basic formulas to prove in the induction cases and the equations defining the operators. The operators and equations are declared in a module, say `ISTEP` (which imports `INV`), as follows:

```
op istepk : Vpk -> Bool
eq istepk(Zk) = invk(s,Zk) implies invk(s',Zk) .
```

for $i = 1, \ldots, N$. `s` and `s'`, which are declared in `ISTEP`, are constants of `H`. `s` denotes an arbitrary state and `s'` denotes a successor state of the state.

The proof plan of (3), written in CafeOBJ, has the form

```
open INV
 red invk(init,zkc) .
close
```

for $i = 1, \ldots, N$. The command `open` makes a temporary module that imports a given module and the command `close` destroys it. The command `red` reduces a given term. CafeOBJ scripts like this constitute proof scores. Such fragments of proof scores are called *proof passages*. Feeding such a proof passage into the CafeOBJ system, if the CafeOBJ system returns `true`, the corresponding proof is successfully done.

The proof of (4) often needs case splitting. We suppose that the state space is split into L_k sub-spaces[4] in order to prove (4) and that each sub-space is characterized by a proposition case_{kl} for $l = 1, \ldots, L_k$ provided that $\mathrm{case}_{k1} \vee \ldots \vee \mathrm{case}_{kL_k}$. The proof of (4) can be then replaced with

$$\begin{array}{l}\mathrm{case}_{kl} \Rightarrow \\ [(P_\alpha(\upsilon^{\mathrm{c}}, d_\alpha) \wedge P_\beta(\upsilon^{\mathrm{c}}, d_\beta) \wedge \ldots) \Rightarrow [P_k(\upsilon^{\mathrm{c}}, z_k^{\mathrm{c}}) \Rightarrow P_k(t_{y_1^{\mathrm{c}},\ldots,y_n^{\mathrm{c}}}(\upsilon^{\mathrm{c}}), z_k^{\mathrm{c}})]]\end{array} \tag{5}$$

for $l = 1, \ldots, L_k$ and $k = 1, \ldots, N$.

We suppose that $\mathtt{d}_\alpha, \mathtt{d}_\beta, \ldots$ are CafeOBJ terms denoting $d_\alpha, d_\beta, \ldots$ Then the proof passage of (5) has the form

```
open ISTEP
 -- arbitrary objects
 op y1c : -> V1 .  ···  op yNc : -> VN .
 -- assumptions
 Declaration of equations denoting casekl.
 -- successor state
 eq s' = t(s,y1c,...,yNc) .
 -- check
 red (invα(s,dα) and invβ(s,dβ) and ...) implies istepk(zkc) .
close
```

[4] Generally, such case splitting should be done while $\forall \upsilon : \mathcal{R}_{\mathcal{S}}.\, p(\upsilon)$ is being proved.

for $l = 1, \ldots, L_k$ and $k = 1, \ldots, N$.

Equations *available in a proof passage* "`open` $M \cdots$ `close`" are those declared in the module M and the modules imported by M plus those declared in the proof passage. We say that the lefthand side of an equation $l = r$ (a term t) is *(ir)reducible in a proof passage* if l (t) is (ir)reducible wrt $E \setminus \{l = r\}$ (E), where E is the set of all equations available in the proof passage.

We briefly describe the proof scores of $\forall \upsilon : \mathcal{R}_{\mathcal{S}_{\mathrm{Tlock}}}.\mathrm{MX}(\upsilon)$.

Example 3 (Proof socres of $\forall \upsilon : \mathcal{R}_{\mathcal{S}_{\mathrm{Tlock}}}.\mathrm{MX}(\upsilon)$). We need four more state predicates to prove $\forall \upsilon : \mathcal{R}_{\mathcal{S}_{\mathrm{Tlock}}}.\mathrm{MX}(\upsilon)$, which are found while proving it. The four state predicates are as follows: $p_2(\upsilon) \triangleq \forall i, j : \mathrm{Pid}.\,[(\mathrm{pc}_i(\upsilon) = \mathrm{cs} \wedge \mathrm{pc}_j(\upsilon) = \mathrm{l2} \wedge \mathrm{ticket}_j(\upsilon) = \mathrm{turn}(\upsilon)) \Rightarrow i = j]$, $p_3(\upsilon) \triangleq \forall i : \mathrm{Pid}.\,(\mathrm{pc}_i(\upsilon) = \mathrm{cs} \Rightarrow \mathrm{turn}(\upsilon) < \mathrm{tvm}(\upsilon))$, $p_4(\upsilon) \triangleq \forall i, j : \mathrm{Pid}.\,[(\mathrm{pc}_i(\upsilon) = \mathrm{l2} \wedge \mathrm{pc}_j(\upsilon) = \mathrm{l2} \wedge \mathrm{ticket}_i(\upsilon) = \mathrm{ticket}_j(\upsilon) \Rightarrow i = j]$, and $p_5(\upsilon) \triangleq \forall i : \mathrm{Pid}.\,(\mathrm{pc}_i(\upsilon) = \mathrm{l2} \Rightarrow \mathrm{ticket}_i(\upsilon) < \mathrm{tvm}(\upsilon))$. The proof of $\forall \upsilon : \mathcal{R}_{\mathcal{S}_{\mathrm{Tlock}}}.\mathrm{MX}(\upsilon)$ needs p_2, that of $\forall \upsilon : \mathcal{R}_{\mathcal{S}_{\mathrm{Tlock}}}.p_2(\upsilon)$ needs MX, p_3 and p_4, that of $\forall \upsilon : \mathcal{R}_{\mathcal{S}_{\mathrm{Tlock}}}.p_3(\upsilon)$ needs MX and p_5, that of $\forall \upsilon : \mathcal{R}_{\mathcal{S}_{\mathrm{Tlock}}}.p_4(\upsilon)$ needs p_5, and that of $\forall \upsilon : \mathcal{R}_{\mathcal{S}_{\mathrm{Tlock}}}.p_5(\upsilon)$ needs no other state predicates.

The module `INV` is declared as follows:

```
mod INV {  pr(TLOCK)
  ops i j : -> Pid
  op inv1 : Sys Pid Pid -> Bool  op inv2 : Sys Pid Pid -> Bool
  op inv3 : Sys Pid -> Bool      op inv4 : Sys Pid Pid -> Bool
  op inv5 : Sys Pid -> Bool
  var S : Sys  vars I J : Pid
  eq inv1(S,I,J) = ((pc(S,I) = cs and pc(S,J) = cs) implies I = J) .
  eq inv2(S,I,J) = ((pc(S,I) = cs and pc(S,J) = l2
                     and ticket(S,J) = turn(S)) implies I = J) .
  eq inv3(S,I)   = (pc(S,I) = cs implies turn(S) < tvm(S)) .
  eq inv4(S,I,J) = ((pc(S,I) = l2 and pc(S,J) = l2
                     and ticket(S,I) = ticket(S,J)) implies I = J) .
  eq inv5(S,I)   = (pc(S,I) = l2 implies ticket(S,I) < tvm(S)) .
}
```

The module `ISTEP` is declared as follows:

```
mod ISTEP {  pr(INV)
  ops s s' : -> Sys
  op istep1 : Pid Pid -> Bool  op istep2 : Pid Pid -> Bool
  op istep3 : Pid -> Bool      op istep4 : Pid Pid -> Bool
  op istep5 : Pid -> Bool
  vars I J : Pid
  eq istep1(I,J) = inv1(s,I,J) implies inv1(s',I,J) .
  eq istep2(I,J) = inv2(s,I,J) implies inv2(s',I,J) .
  eq istep3(I)   = inv3(s,I) implies inv3(s',I) .
  eq istep4(I,J) = inv4(s,I,J) implies inv4(s',I,J) .
  eq istep5(I)   = inv5(s,I) implies inv5(s',I) .
}
```

Let us consider the following proof passage of $\forall v : \mathcal{R}_{\mathcal{S}_{\mathrm{Tlock}}}. \mathrm{MX}(v)$:

```
open ISTEP
-- arbitrary values
  op k : -> Pid .
-- assumptions
  -- eq c-check(s,k) = true .
  eq pc(s,k) = l2 .  eq ticket(s,k) = turn(s) .
  qq i = k .  eq (j = k) = false .  eq pc(s,j) = cs .
-- successor state
  eq s' = check(s,k) .
-- check
  red istep1(i,j) .
close
```

The proof passage corresponds to a (sub-)case obtained by splitting the induction case for check_k. The (sub-)case is referred as case 1.check.1.1.0.1. CafeOBJ returns `false` for the proof passage. From the five equations that characterize the (sub-)case, however, we can conjecture p_2. When `inv2(s,j,i) implies istep1(i,j)` is used instead of `istep1(i,j)`, CafeOBJ returns `true` for the proof passage.

Let us consider the following proof passage of $\forall v : \mathcal{R}_{\mathcal{S}_{\mathrm{Tlock}}}. p_2(v)$:

```
open ISTEP
-- arbitrary values
  op k : -> Pid .
-- assumptions
  -- eq c-exit(s,k) = true .
  eq pc(s,k) = cs .
  eq (i = k) = false .  eq (j = k) = false .  eq pc(s,i) = cs .
-- successor state
  eq s' = exit(s,k) .
-- check
  red istep2(i,j) .
close
```

The proof passage corresponds to a (sub-)case obtained by splitting the induction case for exit_k. The (sub-)case is referred as case 2.exit.1.0.0.1. Although CafeOBJ returns neither `true` nor `false` for the proof passage, we notice that `inv1(s,i,k)` reduces to `false` in the proof passage. Therefore, we use `inv1(s,i,k) implies istep2(i,j)` instead of `istep2(i,j)` and then CafeOBJ returns `true` for the proof passage. □

3 Tips

What we should do to prove a state predicate invariant wrt an OTS is three tasks: (1) use of simultaneous induction, (2) case splitting and (3) predicate (lemma) discovery/use. We use the proof of $\forall v : \mathcal{R}_{\mathcal{S}_{\mathrm{Tlock}}}. \mathrm{MX}(v)$ to describe the three tasks.

3.1 Simultaneous Induction

The first thing to do is to use simultaneous induction to break the proof into the four (sub-)goals (one is the base case and the others are the three induction cases) and the four proof passages are written. The proof passage of the base case is as follows:

```
open INV
  red inv1(init,i,j) .
close
```

The proof passage of the induction case for check_k is as follows:

```
open ISTEP
  op k : -> Pid .
  eq s' = check(s,k) .
  red istep1(i,j) .
close
```

The case is referred as case 1.check. The proof passages of the remaining two induction cases are written likewise.

CafeOBJ returns `true` for the base case but neither `true` nor `false` for each of the three induction cases. What to do for the three induction cases are case splitting and/or predicate discovery/use.

3.2 First Thing to Do for Each Induction Case

Each induction case for $t_{y_1,\ldots,y_n}$ is split into two (sub-)cases: (1) $c\text{-}t_{y_1,\ldots,y_n}$ and (2) $\neg c\text{-}t_{y_1,\ldots,y_n}$ unless $c\text{-}t_{y_1,\ldots,y_n}$ holds in every case. Case 1.check is split into the two (sub-)cases whose corresponding proof passages are as follows:

```
open ISTEP
  op k : -> Pid .
  eq c-check(s,k) = true .
  eq s' = check(s,k) .
  red istep1(i,j) .
close
```

```
open ISTEP
  op k : -> Pid .
  eq c-check(s,k) = false .
  eq s' = check(s,k) .
  red istep1(i,j) .
close
```

The two (sub-)cases are referred as case 1.check.1 and 1.check.0. CafeOBJ returns `true` for case 1.check.0 but neither `true` nor `false` for case 1.check.1. CafeOBJ always returns `true` for the (sub-)case where $\neg c\text{-}t_{y_1,\ldots,y_n}$ due to Definition 1 if the OTS concerned is correctly written in CafeOBJ.

3.3 Appropriate Equations Declared in Proof Passages

As shown, each (sub-)case is characterized by equations. Equational reasoning by rewriting is used to check if a proposition holds in each case, but full equational reasoning power is not used because CafeOBJ does not employ any completion facilities. Therefore, equations that characterize a case heavily affects the success

in proving that a proposition holds in the case. We describe appropriate equations, which characterize a case, declared in a proof passage. If CafeOBJ returns `true` for a proof passage, nothing should be done. Otherwise, the equations in the proof passage should be appropriate as described from now.

- The lefthand side of each equation should be irreducible in a proof passage so that the equation can be used effectively as a rewrite rule. This is because the rewriting strategy adopted by CafeOBJ is basically an innermost strategy.
- Let $PP(E)$, where E is a set of equations, be a proof passage in which the equations in E are declared, and E_1 and E_2 be sets of equations. We suppose that $\bigwedge_{e_1 \in E_1} e_1$ is equivalent to $\bigwedge_{e_2 \in E_2} e_2$. If every equation in E_1 can be proved by rewriting from $PP(E_2)$ but every equation in E_2 cannot be proved by rewriting from $PP(E_1)$, then E_2 should be used instead of E_1. Some examples are given.
  1. Let E_1 be $\{\rho_1 \wedge \rho_2 = \texttt{true}\}$ and E_2 be $\{\rho_1 = \texttt{true}, \rho_2 = \texttt{true}\}$. We suppose that $\rho_1 \wedge \rho_2$, ρ_1 and ρ_2 are irreducible in $PP(\emptyset)$. Then, $\rho_1 \wedge \rho_2$ reduces to `true` in $PP(E_2)$ but l_1 (l_2) does not necessarily reduce to `true` in $PP(E_1)$. Therefore, E_2 should be used instead of E_1.
  2. Let c be a binary data constructor. We suppose that $c(a_1, b_1)$ equals $c(a_2, b_2)$ if and only if a_1 equals a_2 and b_1 equals b_2. Let E_1 be $\{c(a_1, b_1) = c(a_2, b_2)\}$ and E_2 be $\{a_1 = a_2, b_1 = b_2\}$. We suppose that $c(a_1, b_1)$, a_1 and b_1 are irreducible in $PP(\emptyset)$. Then, both $c(a_1, b_1)$ and $c(a_2, b_2)$ reduce to a same term in $PP(E_2)$ but a_1 and a_2 (b_1 and b_2) do not necessarily reduce to a same term in $PP(E_1)$. Therefore, E_2 should be used instead of E_1.
  3. Let n be a natural number, N be a constant denoting an arbitrary multiset of natural numbers, the juxtaposition operator be a data constructor of multisets. The juxtaposition operator is declared as `op __ : Bag Bag -> Bag {assoc comm id: empty}`, where `Bag` is the visible sort for multisets of natural numbers and is a supersort of `Nat`, `assoc` and `comm` specify that the operator is associative and commutative, and `id: empty` specifies that `empty`, which is the constant denoting the empty multiset, is an identity of the operator. We suppose that we want to specify that N includes n. One way is to use $n \in N = \texttt{true}$, and the other way is to use $N = n\ N'$, where N' is another constant denoting an arbitrary multiset of natural numbers[5]. Let E_1 be $\{n \in N = \texttt{true}\}$ and E_2 be $\{N = n\ N'\}$. We suppose that $n \in N$ and N are irreducible in $PP(\emptyset)$. Then, $n \in N$ reduces to `true` in $PP(E_2)$ if $\in$ is defined appropriately in equation, but N and $n\ N'$ do not necessarily reduce to a same term in $PP(E_1)$. Therefore, E_2 should be used instead of E_1.
  4. $\neg\rho$ is reducible in any proof passage because of the Hsiang TRS. If ρ is irreducible in a proof passage, $\neg\rho$ reduces to ρ xor `true` in the proof

[5] Since N is an arbitrary multiset and includes n, N must be $n'\ N'$, where (1) n' equals n or (2) $n \in N'$. We can select (1) because the juxtaposition operator is associative and commutative.

passage. Therefore, one way of making the equation $(\neg\rho) = \texttt{true}$ effective is to use $(\rho$ xor $\texttt{true}) = \texttt{true}$. But, $\rho = \texttt{false}$ is more appropriate.

5. This example is a variant. Let E_1 be $\{(l = r) = \texttt{true}\}$ and E_2 be $\{l = r\}$. We suppose that $l = r$ and l are irreducible in $PP(\emptyset)$. $l = r$ reduces to `true` in both $PP(E_1)$ and $PP(E_2)$. It is often the case, however, that E_2 is more appropriate than E_1 because l reduces r in $PP(E_2)$ but l does not in $PP(E_1)$.

According to what has been described in this subsection, the proof passage of case 1.check.1 should be rewritten as follows:

```
open ISTEP
  op k : -> Pid .
  -- eq c-check(s,k) = true .
  eq pc(s,k) = l2 .  eq ticket(s,k) = turn(s) .
  eq s' = check(s,k) .
  red istep1(i,j) .
close
```

CafeOBJ still returns neither `true` nor `false` for this proof passage. Then, what we should do is further case splitting.

3.4 Further Case Splitting

For a proof passage for which CafeOBJ returns neither `true` nor `false`, the case corresponding to the proof passage is split into multiple (sub-)cases in each of which CafeOBJ returns either `true` or `false`. When CafeOBJ returns `true` in a (sub-)case, nothing should be done for the case. When CafeOBJ returns `false` in a (sub-)case, it is necessary to find a state predicate that does not hold in the case and is likely invariant wrt an OTS concerned.

There are some ways of splitting a case into multiple (sub-)cases.

- Based on a proposition ρ: A case is split into two (sub-)cases where (1) ρ holds and (2) ρ does not, respectively. As shown in Subsect. 3.2, case 1.check is split into the two (sub-)cases based on the proposition `c-check(s,k)`.
- Based on data constructors: We suppose that a data type has M data constructors. Then, a case is split into M (sub-)cases. Some examples are given.
  1. `Nat` has the two data constructors `0` and `s`. Let x be a constant denoting an arbitrary natural number in a proof passage. The case corresponding to the proof passage is split into the two (sub-)cases where (1) $x = \texttt{0}$ and (2) $x = \texttt{s}(y)$, where y is another constant denoting an arbitrary natural number. Case (1) means that x is zero and case (2) means that x is not zero.
  2. `Bag` has the two data constructors `empty` and `__`. Let N be a constant denoting an arbitrary multiset in a proof passage. The case corresponding to the proof passage is split into the two (sub-)cases where (1) $N = \texttt{empty}$ and (2) $N = n'\ N'$, where n' is a constant denoting an arbitrary natural number and N' is a constant denoting an arbitrary multiset. Case (1) means that N is empty and case (2) means that N is not empty.

– Based on a tautology whose form is $\rho_1 \vee \ldots \vee \rho_M$: A case is split into M (sub-)cases where (1) ρ_1 holds, ..., (M) ρ_M holds. This case splitting generalizes the case splitting based on a proposition because $\rho \vee \neg\rho$ is a tautology.

In order to apply one of the three ways of splitting a case, we need to find a proposition, a constant denoting an arbitrary value of a data type, or a tautology whose form is $\rho_1 \vee \ldots \vee \rho_M$. There are usually multiple candidates based on which a case is split. A selection from such candidates affects how well a proof concerned is conducted. It is necessary to understand an OTS concerned and experience writing proof scores so as to select a better one among such candidates. There are some heuristic rules, however, to select one among such candidates.

– Select a proposition that directly affects the truth value of a proposition to prove such as `istep(i,j)`. If `i` equals `j`, `istep(i,j)` reduces to `true` in case 1.check.1, the proposition `i = j` may be a good candidate.
– Select a proposition ρ if ρ appears in a result obtained by reducing a proposition to prove. If ρ appears at the conditional position of `if_then_else_fi` such as `if` ρ `then` a `else` b `fi`, ρ may be a good candidate.

We describe how to split case 1.check.1. CafeOBJ returns `((if (k = i) then cs else pc(s,i) fi) = cs) and ...` for the corresponding proof passage. Then, we select the proposition `k = i` to split the case. The equation `i = k` is declared[6] in one proof passage whose corresponding case is referred as case 1.check.1.1, and the equation `(i = k) = false` is declared in the other proof passage whose corresponding case is referred as case 1.check.1.0.

Since CafeOBJ returns `if (k = j) then cs else pc(s,j) fi = cs and ...` for the proof passage corresponding case 1.check.1.1, we select the proposition `k = j` to split the case. The equation `j = k` is declared in one proof passage whose corresponding case is referred as 1.check.1.1.1, and the equation `(j = k) = false` is declared in the other proof passage whose corresponding case is referred as case 1.check.1.1.0. CafeOBJ returns `true` for the former proof passage, but `pc(s,j) = cs xor true` for the latter proof passage. Then, case 1.check.1.1.0 is also split based on `pc(s,j) = cs`. The equation `pc(s,j) = cs` is declared in one proof passage whose corresponding case is referred as case 1.check.1.1.0.1, and the equation `(pc(s,j) = cs) = false` is declared in the other proof passage whose corresponding case is referred as case 1.check.1.1.0.0. CafeOBJ returns `false` for the former proof passage and `true` for the latter proof passage. Case 1.check.1.0 can be split into four (sub-)cases in the same was as case 1.check1.1.

3.5 Predicate (Lemma) Discovery/Use

When CafeOBJ returns `false` for a proof passage, there are two possibilities: (1) if an an arbitrary state characterized by the case corresponding to the proof passage is not reachable wrt an OTS $\mathcal{S}$ concerned, the case can be discharged,

[6] Note that `i = k` is declared instead for `k = i`.

and (2) otherwise, a state predicate concerned is not invariant wrt $\mathcal{S}$. If a state predicate is invariant wrt $\mathcal{S}$ and does not hold in the case, then an arbitrary state characterized by the case is not reachable wrt $\mathcal{S}$. That is why we find a state predicate that does not hold in the case and is likely invariant wrt $\mathcal{S}$.

Let E is a set of equations that characterize a case such that CafeOBJ returns `false` for a proof passage corresponding to the case. We suppose that $\bigwedge_{e \in E} e$ is equivalent to a proposition whose form is $Q(\upsilon^{\mathrm{c}}, z_{\alpha}^{\mathrm{c}})$. Let $q(\upsilon)$ be $\forall z_{\alpha} : D_{q\alpha}. \neg Q(\upsilon, z_{\alpha})$. Since q surely does not hold in the case characterized by E, q is one possible candidate. Generally, q' such that $q' \Rightarrow q$ can be a candidate because q' does not hold in the case characterized by E,

Let us consider the proof passage corresponding to case 1.check.1.1.0.1 shown in Example 3. From the five equations that characterize the case, we obtain the proposition `pc(s,i) = l2 and pc(s,j) = cs and ticket(s,i) = turn(s) and not(j = i)` by concatenating them with conjunctions, substituting `k` with `i` because of the equation `i = k`, and deleting the tautology `i = i`. p_2 is obtained from the proposition,

Some contradiction may be found in a set of equations that characterize a case even when CafeOBJ does not return `false` in a proof passage corresponding to the case. If that is the case, a state predicate can be obtained from the contradiction such that the state predicate does not hold in the case and is likely invariant wrt an OTS concerned.

Let us consider the proof passage corresponding to case 2.exit.1.0.0.1 shown in Example 3. We notice that the three equations `pc(s,k) = cs`, `pc(s,i) = cs` and `(i = k) = false` contradict $\forall \upsilon : \mathcal{R}_{\mathcal{S}_{\mathrm{Tlock}}}. \mathrm{MX}(\upsilon)$ and `inv1(s,i,k)` can be used in the proof passage.

Even when any contradictions are not found in a set of equations that characterize a case and CafeOBJ does not return `false` in a proof passage corresponding to the case, a state predicate may be found such that the state predicate can be used to discharge the case and is likely invariant wrt an OTS concerned.

Let us consider the proof passage corresponding to case 1.check.1.1.0. CafeOBJ returns `pc(s,j) = cs xor true` for the proof passage, but `inv2(s,j,i)` also reduces to `pc(s,j) = cs xor true` in the proof passage. Therefore, `inv2(s,j,i)` can be used to discharge the case and it is not necessary to split the case anymore.

4 Soundness of Proof Scores

Let us consider the proof of $\forall \upsilon : \mathcal{R}_{\mathcal{S}}. (p_1(\upsilon) \wedge \ldots \wedge p_N(\upsilon))$ described in Subsect. 2.4 again. If CafeOBJ returns `true` for each proof passage in the proof scores, $p_1, \ldots, p_N$ are really invariant wrt $\mathcal{S}$ provided that

1. Needless to say, the computer (including the operating system, the hardware, etc.) on which CafeOBJ works is reliable,
2. Equational reasoning is sound and rewriting faithfully (partially though) implements equational reasoning [8]; the CafeOBJ implementation of rewriting is reliable,

3. The Hsiang TRS is sound [12]; the TRS is reliably implemented in CafeOBJ,
4. The built-in equality operator `_==_` is not used,
5. $\mathcal{S}$ is specified in CafeOBJ in the way described in Subsect. 2.3, and
6. The proof scores of $\forall v : \mathcal{R}_{\mathcal{S}}.\, (p_1(v) \wedge \ldots \wedge p_N(v))$ are written in the way described in Subsect. 2.4.

When CafeOBJ meets the term a `==` b, it first reduces a and b to a' and b', which are irreducible wrt a set of equations (rewrite rules) concerned, and returns `true` if a' is exactly the same as b' and `false` otherwise. The combination of `_==_` and `not_` can damage the soundness. Since the built-in inequality operator `_=/=_` is the combination of `_==_` and `not_`, it should not be used either. Let us consider the following module:

```
mod! DATA { [Data]
  ops d1 d2 : -> Data
}
```

We try to prove $\forall d : \texttt{Data}.\, \neg(d = \texttt{d2})$ by writing a proof score. A plausible proof score that consists of one proof passage is as follows:

```
open DATA
  op d : -> Data .   -- an arbitrary value of Data.
  red not(d == d2) . -- or red d =/= d2 .
close
```

CafeOBJ returns `true` for this proof passage, which contradicts the fact that there exists the counterexample `d2`. Therefore, users should declare an equality operator such as `_=_` for each visible sort and equations defining it instead of `_==_` and `_=/=_`.

Under the above six assumptions, the only thing that we should take care of on the soundness is whether all necessary cases are checked by rewriting for each proof passage. A possible source of damaging it is transitions. Since transitions are functions on states in OTSs, however, the source can be dismissed. Every operator is a function in CafeOBJ as well. Therefore, rewriting surely covers all necessary cases for each proof passage.

Note that we do not have to assume that the CafeOBJ specification of $\mathcal{S}$, when it is regarded as a TRS, is terminating or confluent for the soundness. If the CafeOBJ specification is not terminating, CafeOBJ may not return any results for a proof passage forever. This causes the success in proofs, but does not affect the soundness.

We suppose that a term a has two irreducible forms a' and a'' in a proof passage because the CafeOBJ specification is not confluent and that a actually reduces to a' but not to a''. Although CafeOBJ ignores a rewriting sequence that starts with a and ends in a'', this does not affect the soundness because a' equals a'' from an equational reasoning point of view and it is enough to use either a' or a''. Whether the CafeOBJ specification is confluent, however, can affects the success in proofs. Let us consider the following module:

```
mod! DATA2 { [Data2]
  ops d1 d2 d3 : -> Data2
  op _=_ : Data2 Data2 -> Bool {comm}
  var D : Data2
  eq (D = D) = true .
  eq d1 = d2 .  eq d1 = d3 .
}
```

We try to prove `d1` = `d3` by writing a proof passage. The case is split into two (sub-)cases where (1) `d2` = `d3` and (2) `d2` ≠ `d3`. Then, the proof score that consists of two proof passages is as follows:

```
open DATA2
  eq d2 = d3 .
  red d1 = d3 .
close
open DATA2
  eq (d2 = d3) = false .
  red d1 = d3 .
close
```

CafeOBJ returns `true` for the first proof passage and `false` for the second proof passage. We stuck for the second proof passage unless we notice the equation `d1 = d3` in the module `DATA2`.

From what has been described, it is desirable that the CafeOBJ specification of $\mathcal{S}$ is terminating and confluent.

We can check if proof scores that state predicates are invariant wrt $\mathcal{S}$ conforms to what is described in Subsect. 2.4. We suppose that all proofs are conducted by simultaneous induction. Let P and P' be sets of state predicate such that P' is empty. A procedure that makes such a check is as follows:

1. If P is empty, the procedure successfully terminates, which means that the proof score of $\forall v : \mathcal{R}_{\mathcal{S}}.\, p(v)$ for each $p \in P'$ conforms to what is described in Subsect. 2.4; otherwise, extract a predicate p from P and go next.
2. Check if a proof score of $\forall v : \mathcal{R}_{\mathcal{S}}.\, p(v)q$ has been written. If so, go next; otherwise, the procedure reports that a proof score of $\forall v : \mathcal{R}_{\mathcal{S}}.\, p(v)$ has not been written and terminates.
3. Check if the proof score of $\forall v : \mathcal{R}_{\mathcal{S}}.\, p(v)q$ conforms to simultaneous induction. If so, go next; otherwise, the procedure reports that the proof score of $\forall v : \mathcal{R}_{\mathcal{S}}.\, p(v)q$ does not conform to simultaneous induction and terminates.
4. Check if the proof score of $\forall v : \mathcal{R}_{\mathcal{S}}.\, p(v)q$ covers all necessary cases. If so, put p into P', put other state predicates that are used in the proof score and that are not in P' into P, and go to 1; otherwise, the procedure reports that the proof score of $\forall v : \mathcal{R}_{\mathcal{S}}.\, p(v)$ does not cover all necessary cases and terminates.

The procedure can increase the confidence in soundness of proof scores.

5 Conclusion

We have described some tips on writing proof scores in the OTS/CafeOBJ method and used Tlock, a mutual exclusion protocol using *atomicInc*, to exemplify the tips. We have also informally argued soundness of proof scores in the OTS/CafeOBJ method.

We have been developing a tool called Gateau [21] that takes propositions used for case splitting and state predicates used to strengthen the basic induction hypothesis, and generates the proof score of an invariant, which conforms to what is described in Subsect. 2.4.

Proof scores can also be considered proof objects, which can be checked as described in Sect. 4. We think that it is worthwhile to develop a tool, which is an implementation of the procedure in Sect. 4 that checks if a proof score conforms to what is described in Subsect. 2.4. Such a tool can be complementary to Gateau.

References

1. Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development – Coq'Art: The Calculus of Inductive Constructions*. Springer, 2004.
2. R. Diaconescu and K. Futatsugi. *CafeOBJ Report*, volume 6 of *AMAST Series in Computing*. World Scientific, 1998.
3. R. Diaconescu and K. Futatsugi. Behavioural coherence in object-oriented algebraic specification. *J. UCS*, 6:74–96, 2000.
4. R. Diaconescu, K. Futatsugi, and K. Ogata. CafeOBJ: Logical foundations and methodologies. *Computing and Informatics*, 22:257–283, 2003.
5. K. Futatsugi, J. A. Goguen, J. P. Jouannaud, and J. Meseguer. Principles of OBJ2. In *12th POPL*, pages 52–66. ACM, 1985.
6. K. Futatsugi, J. A. Goguen, and K. Ogata. Verifying design with proof scores. In *VSTTE 2005*, 2005.
7. K. Futatsugi, J. A. Goguen, and K. Ogata. Formal verification with the OTS/CafeOBJ method. submitted for publication, 2006.
8. J. Goguen. *Theorem Proving and Algebra*. The MIT Press, to appear.
9. J. Goguen and G. Malcolm. A hidden agenda. *TCS*, 245:55–101, 2000.
10. J. Goguen and G. Malcolm, editors. *Software Engineering with OBJ: Algebraic Specification in Action*. Kluwer, 2000.
11. J. V. Guttag, J. J. Horning, S. J. Garland, K. D. Jones, A. Modet, and J. M. Wing. *Larch: Languages and Tools for Formal Specification*. Springer, 1993.
12. J. Hsiang and N. Dershowitz. Rewrite methods for clausal and nonclausal theorem proving. In *10th ICALP*, LNCS 154, pages 331–346. Springer, 1983.
13. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. LNCS 2283. Springer, Berlin, 2002.
14. K. Ogata and K. Futatsugi. Flaw and modification of the *i*KP electronic payment protocols. *IPL*, 86:57–62, 2003.
15. K. Ogata and K. Futatsugi. Formal analysis of the *i*KP electronic payment protocols. In *1st ISSS*, LNCS 2609, pages 441–460. Springer, 2003.
16. K. Ogata and K. Futatsugi. Formal verification of the Horn-Preneel micropayment protocol. In *4th VMCAI*, LNCS 2575, pages 238–252. Springer, 2003.

17. K. Ogata and K. Futatsugi. Proof scores in the OTS/CafeOBJ method. In *6th FMOODS*, LNCS 2884, pages 170–184. Springer, 2003.
18. K. Ogata and K. Futatsugi. Equational approach to formal verification of SET. In *4th QSIC*, pages 50–59. IEEE CS Press, 2004.
19. K. Ogata and K. Futatsugi. Formal analysis of the NetBill electronic commerce protocol. In *2nd ISSS*, volume 3233 of *LNCS*, pages 45–64. Springer, 2004.
20. K. Ogata and K. Futatsugi. Equational approach to formal analysis of TLS. In *25th ICDCS*, pages 795–804. IEEE CS Press, 2005.
21. T. Seino, K. Ogata, and K. Futatsugi. A toolkit for generating and displaying proof scores in the OTS/CafeOBJ method. In *6th RULE*, ENTCS 147(1), pages 57–72. Elsevier, 2006.