Advances of Proof Scores in CafeOBJ

Kokichi Futatsugi

Japan Advanced Institute of Science and Technology (JAIST),
Nomi, Ishikawa, 923-1292, Japan

Abstract

Critical flaws continue to exist at the level of domain, requirement, and/or design specification, and specification verification (i.e., to check whether a specification has desirable properties) is still one of the most important challenges in software/system engineering. CafeOBJ is an executable algebraic specification language system and domain/requirement/design engineers can write proof scores for improving quality of specifications by the specification verification. This paper describes advances of the proof scores for the specification verification in CafeOBJ.

Keywords: Formal/Algebraic Specifications; Specification Verification and Validation; Interactive Theorem Proving; Proof Scores; CafeOBJ

1. Introduction

Specification verification is theorem proving where a specification is a set of axioms and desirable properties are theorems. The main goal of a specification verification is not, however, to prove a property holds for the specification but to check the quality of the specification against the property; a specification is supposed to improve via the specification verification. This is quite different from ordinary theorem proving where the set of axioms is fixed and not supposed to change. The improvements of a specification include the addition of necessary axioms, the deletion of unnecessary axioms, and the improvement of the specification’s module structure.

For achieving the improvements, specifications and the machinery of the specification verification are better to be clear and transparent. That is, the followings are better to be clear and precisely defined, and hopefully simple and transparent. (i) Models of a specification (models of a system specified), (ii) inference (deduction, proof) rules used, and (iii) rationale for assumptions used (i.e., why an assumption is justified to be used or not).

CafeOBJ [6, 10, 11, 21] is an executable algebraic specification language system based on equational logic and rewriting execution with transparent and
precise definition of the models and the inference execution rules. CafeOBJ’s powerful module system makes it possible to describe the assumption conclusion relations in a clear and transparent way.

A fully automatic specification verification often fails to convey important structures properties of the systems specified, and makes it difficult to improve the specification. One should seek to make balanced optimal use of the respective abilities of humans and computers, so that computers do the tedious formal calculations, and humans do the high level planning. Proof scores have been fostered in the OBJ [10, 28, 30, 43], CafeOBJ community, and intend to meet these requirements [12, 13, 20].

A proof score for a specification in CafeOBJ is a piece of code that constructs proof modules and does rewrites reductions on them with the equations (i.e., axioms) in the specification and the proof modules [15, 18] (see Sections 3, 4 for more details). If all the reductions return the expected results, a proof attempted is achieved. The formal calculations are encapsulated into the reductions with the equations, and the high level planning is encoded into the proof modules and the reduction commands on them.

Major high level planning includes lemma case-split, and induction. They are supposed to be found by humans and declared as proof modules by making use of CafeOBJ’s powerful module system. Proof modules can also be constructed with fairly flexible open close constructs (see Section 8), and case-split induction can be described in a liberal way. It involves, however, risks of obscuring the rationale of the high level planning. Proof tree calculus (PTcalc) and well-founded induction (WFI) are incorporated to solve the issue, by declaring a case-split with exhaustive equations and an induction via a well-founded relation [15, 16].

Transition system is the primary modeling scheme for software related dynamic systems. A proof score method making use of CafeOBJ’s builtin search predicates is developed for proving leads-to properties of transition systems [14]. The leads-to properties cover fairly large part of important properties of dynamic systems, and their proof scores can be built with PTcalc+WFI.

This paper is intended to be a comprehensive and self-contained account on the recent advances of the proof scores in CafeOBJ with a nice harmony of theories and cases. Many papers have been published on CafeOBJ and/or proof scores. Majority of them, however, are on theories methods or examples case studies intensively, and few treat the both in valance and describe how theories justify cases in CafeOBJ in a concrete fashion.

For achieving the intention, (1) theories are described so as to justify cases in CafeOBJ as directly as possible, (2) CafeOBJ’s cases are presented to explain the meaning intention of theories in concrete ways. As a result, the paper clarifies the following two things in a fairly simple and transparent way, because the logics the CafeOBJ’s proof scores based are the simplest among ones on which other theorem proving systems based. (a) A minimum cohesive collection of theories that makes proof scores work. (b) Typical cases of proof scores in CafeOBJ that are justified by the theories.
The following sections are organized as follows: Section 2 describes theories that are necessary for justifying proof scores. Section 3 explains how to construct basic proof scores with \texttt{open} \ldots \texttt{close} constructs. Section 4 explains how to construct advanced proof scores with proof tree calculus (PTcalc) and well-founded induction (WFI). Section 5 describes theories and methods for constructing proof scores for transition systems. Section 6 demonstrates the constructions of the PTcalc+WFI proof scores for a simple example of transition system. Section 7 discusses related works, distinctive features of the proof scores, and concludes the paper.

2. Fundamental Theories

This section is a preliminary to the following sections, and is better to be browsed through on first reading. There are many pointers from the following sections to related parts in this section, and readers are encouraged to return to understand necessary concepts and notations.

CafeOBJ supports several kinds of specifications such as equational specifications, rewriting specifications, and behavioral specifications. This paper focuses on equational specifications and transition specifications (a restricted class of rewriting specifications) on which our proof score method has been mainly developed. We will proceed with the following items for CafeOBJ following the style of \cite{10,18}.

- a set $\textit{Sign}$ of \textbf{signatures}.
- for each signature $\Sigma \in \textit{Sign}$ a class $\textit{Mod}(\Sigma)$ of $\Sigma$-\textbf{models}.
- for each signature $\Sigma$ a set $\textit{Sen}(\Sigma)$ of $\Sigma$-\textbf{sentences}.
- for each signature $\Sigma$ a \textbf{satisfaction relation} $\vdash_{\Sigma}$ between $\Sigma$-models and $\Sigma$-sentences, i.e., $\vdash_{\Sigma} \subseteq (\textit{Mod}(\Sigma) \times \textit{Sen}(\Sigma))$.

2.1. Signatures

A signature determines the syntax of terms (or expressions) for describing specifications (or modules of CafeOBJ).

2.1.1. Sorts

A \textbf{sort} is a name for entities of the same kind/sort/type, and interpreted as the set of the entities. A \textbf{subsort} relation can be declared among sorts and is interpreted as a subset relation.

Let $sl = s_1 \cdots s_n$ be a list of names $s_i$ separated by blanks. The CafeOBJ code (i.e., a piece of text in CafeOBJ code) $[sl]$ declares each $s_i$ ($1 \leq i \leq n$) is a sort. Let $sl_1 = s_1 \cdots s_1 m$ and $sl_2 = s_2 \cdots s_2 n$ be two lists of

\footnote{This paper is based on \cite{17}. Section 6 is totally new. All of the other sections are significantly revised and extended.}
names. The code \([sl_1 < sl_2]\) declares that (i) each of \(sl_1(1 \leq i \leq m)\) and \(sl_2(1 \leq j \leq n)\) is a sort, and (ii) \(sl_1\) is a subsort of \(sl_2\) for each pair \((i,j)\) \((1 \leq i \leq m, 1 \leq j \leq n)\) (see Examples \[1\] \[4\]). Let \(sl_i(1 \leq i \leq n)\) be \(n\) lists of names. The code \([sl_1 < sl_2 < \cdots < sl_n]\) \((2 \leq n)\) is equivalent to “\([sl_1 < sl_2]\) \([sl_2 < sl_3] \cdots [sl_{n-1} < sl_n]\)”. Note that CafeOBJ code is written in the type-writer font.

The subsort declarations define the partially ordered set \((\text{poset}) \(S, \leq\)\) where \(S\) is the set of sorts involved and \(\leq\) is the minimum partial order relation including the subsort relations declared. If \(s \leq s'\), \(s\) is called a subsort of \(s'\) and \(s'\) is called a supersort of \(s\). Given a poset \((\text{poset}) \(S, \leq\)\) of sorts, let \(\equiv\) denote the smallest equivalence relation including the partial order \(\leq\). The quotient of \(S\) by the equivalence relation \(\equiv\) is denoted as \(\hat{S} = S/\equiv\), and an element of \(\hat{S}\) is called a connected component.

2.1.2. Operators

An operator (or function) \(f\) is declared as “\(\text{op } f : w \rightarrow s \) .” in CafeOBJ, where \(w \in S^*\) is its arity and \(s \in S\) is its sort (or co-arity) of the operator (see Example \[4\]). The string \(ws\) is called the rank of the operator. A constant \(f\) is an operator whose arity is empty (i.e., \(\text{op } f : \rightarrow s \) ). Let \(F_{ws}\) denote the set of all operators of rank \(ws\), then the whole collection \(F\) of operators can be represented as the family of the sets of operators sorted by (or indexed by) ranks as \(F = \{F_{ws}\}_{w \in S^*, s \in S}\). Note that “\(\text{op } f : w \rightarrow s \) .” iff (if and only if) \(f \in F_{ws}\). Operators can be overloaded, that is the same name can be used for two operators of different ranks. In other words, \(F_{ws}\) and \(F_{w's'}\) can have a common element for different \(ws\) and \(w's'\).

2.1.3. Builtin BOOL and Predicates

CafeOBJ has a builtin module BOOL realizing the Boolean algebra with the sort BOOL and the following operator: \(\text{true, false, not, and, or, xor, implies, iff, _}\) where \(_\) indicates the position of argument. An operator \(f\) with the rank “\(w \text{BOOL}\)” (i.e., with the arity \(w\) and the co-arity BOOL) is called predicate, which can be declared as “\(\text{pred } f : w \) .”.

There is an important builtin equality predicate “\(\text{pred } =_\_ : *\text{Cosmos}* *\text{Cosmos }* \rightarrow \text{BOOL }\)” with equations “\(\text{eq } (\text{CUX}:*\text{Cosmos }* = \text{CUX}) = \text{true }\)” and “\(\text{eq } (\text{true } = \text{false}) = \text{false }\)”, where \(*\text{Cosmos}\) is a builtin sort variable that can be interpreted as any sort Srt. That is, “\(\text{pred } =_\_ : \text{Srt Srt } \rightarrow \text{BOOL}\) .” and “\(\text{eq } (\text{CUX:Srt } = \text{CUX}) = \text{true }\)” is declared for any sort Srt.

The builtin module BOOL is used in a significant way as follows.

* The condition of an equation (see \[2.3.1\]) is a Boolean term.
* A property of interest on a specification is expressed with the predicates.

---

\[\text{Note: Numbers in the italic font delimited by \("\) as 2.3 or 2.3.1 denote a subsection or a subsubsection of this paper.}\]
The semantics (i.e., model) of the builtin BOOL is the standard propositional calculus and is preserved by models of every module (i.e., specification).

2.1.4. Operator Attributes

CafeOBJ supports the following three attributes for a binary operator f.

- **associativity** \{assoc\}: \( f(x, f(y, z)) = f(f(x, y), z) \)
- **commutativity** \{comm\}: \( f(x, y) = f(y, x) \)
- **identity** \{id\}: \( f(x, e) = f(e, x) = x \) (e is an identity constant)

Identity equations are implemented with rewrite rules, and associative-commutative equations are implemented with the matching modulo associativity/commutativity (see 2.6.3).

2.1.5. Order-Sorted Signatures

An order-sorted signature is defined by a tuple \((S, \leq, F)\) (see 2.1.1, 2.1.2).

For making construction of symbolic presentations of models (i.e., term algebras, see 2.2.3) possible, the following sensibility condition is the most general sufficient condition for avoiding ambiguity found until now [36]. An order-sorted signature \((S, \leq, F)\) is defined to be **sensible** iff \((w \equiv \leq w' \Rightarrow s \equiv \leq s')\) for any operator \(f \in F_{ws} \cap F_{w's}\), where \(w \equiv \leq w'\) means that (i) \(w\) and \(w'\) are of the same length and (ii) any element of \(w\) is in the same connected component with corresponding element of \(w'\). Note that \([\_\_] \equiv \leq [\_\_]\) for the empty arity \([\_\_]\). A sensible signature guarantees a unique interpretation of each \(\Sigma(X)\)-term (2.2.3).

An order-sorted signature \((S, \leq, F)\) is defined to be **regular** iff there is a unique least element in the set \(\{ ws | f \in F_{ws} \land w_0 \leq w \}\) for each \(f \in F_{w_1 s_1}\) and each \(w_0 \leq w_1\). A regular signature guarantees that each term has the unique minimum parse (2.2.3).

Each order-sorted signature \((S, \leq, F)\) is assumed to be sensible and regular throughout this paper. By using CafeOBJ’s check commands, sensibility and regularity of a signature can be checked automatically.

**Example 1.** (i) \{[Bool Nat] op 0 : -> Bool . op 0 : -> Nat .\} defines a non sensible signature, and 0 cannot be identified with any entity of any sort.

(ii) \{[Zero < Nat EvenInt] op 2 : -> Nat . op 2 : -> EvenInt .\} defines a sensible but non regular signature, and 2 is identified with an entity that belongs to Nat and EvenInt but it has no minimum parse.

(iii) \{[EvenNat < Nat EvenInt] op 2 : -> EvenNat . op 2 : -> Nat . op 2 : -> EvenInt .\} defines a sensible and regular signature, and 2 is identified with the entity that belongs to EvenNat, Nat, and EvenInt with the minimum parse 2:EvenNat.

2.1.6. Constructor-Based Order-Sorted Signatures

A **constructor-based order-sorted signature** \((S, \leq, F, F^c)\) is an order-sorted signature with constructor declarations. \(F^c \subseteq F\) is a distinguished
subfamily of sets of operators, called constructors, such that \( (S, \leq, F) \) and \( (S, \leq, F^c) \) are order-sorted signatures. Constructors represent data on which a specification is constructed, and non-constructors represent functions over the data (see Example 4). An operator \( f \in F^c_{w s} \) or \( f \in F^c_{w s'} \) with \( s' \leq s \) is called a constructor of sort \( s \). A sort \( s \in S \) is called constrained if there exists a constructor of \( s \). Let \( S^c \) denote the set of all constrained sorts. An element of \( S^c \) is called a loose sort.

2.2. Models

A CafeOBJ’s signature \( \Sigma \) determines the set \( \text{Mod}(\Sigma) \) of models called \( \Sigma \)-algebras. An algebra is a collection of sorted sets with operators (functions) on the sets.

2.2.1. \( (S, \leq, F) \)-Algebras

Let \( (S, \leq, F) \) be an order-sorted signature. An \( (S, \leq, F) \)-algebra (or an order-sorted algebra of signature \( (S, \leq, F) \)) \( A \) interprets (i) each sort \( s \in S \) as a carrier set \( A_s \), (ii) each subset relation \( s < s' \) as an inclusion \( A_s \subseteq A_{s'} \), and (iii) each operator \( f \in F_{s_1...s_n,s} \) as a function \( A_f : A_{s_1} \times \ldots \times A_{s_n} \rightarrow A_s \) such that any two functions of the same name return the same value if applied to the same argument, i.e., if \( f : w \rightarrow s \) and \( f : w' \rightarrow s' \) and \( ws \equiv ws' \) and \( \bar{\pi} \in A_w \cap A_{w'} \) then \( A_{f:w \rightarrow s}(\bar{\pi}) = A_{f:w' \rightarrow s'}(\bar{\pi}) \).

2.2.2. \( (S, \leq, F) \)-Algebra Morphisms

Let \( A, B \) be \( (S, \leq, F) \)-algebras. An \( (S, \leq, F) \)-algebra morphism (or model morphism) \( h : A \rightarrow B \) is an \( S \)-sorted family of functions between the carrier sets of \( A \) and \( B \), \( \{h_s : A_s \rightarrow B_s\}_{s \in S} \), such that

- \( h_s(A_f(a_1, \ldots, a_n)) = B_f(h_{s_1}(a_1), \ldots, h_{s_n}(a_n)) \) for all \( f \in F_{s_1...s_n,s} \) (0 \( \leq n \)) and \( a_i \in A_{s_i} \) for \( i \in \{1, \ldots, n\} \), and
- if \( s \equiv s' \) and \( a \in A_s \cap A_{s'} \) then \( h_s(a) = h_{s'}(a) \).

2.2.3. Terms and Term Algebras

Let \( \Sigma = (S, \leq, F) \) be an order-sorted signature, and \( X = \{X_s\}_{s \in S} \) be an \( S \)-sorted, mutually disjoint countably infinite, sets of variables called a \( \Sigma \)-variable set. \( \Sigma(X) \)-term is defined inductively as follows.

- Each constant \( f \in F_s \) is a \( \Sigma(X) \)-term of sort \( s \).
- Each variable \( x \in X_s \) is a \( \Sigma(X) \)-term of sort \( s \).
- \( t \) is a term of sort \( s' \) if \( t \) is a term of sort \( s \) and \( s < s' \).
- \( f(t_1, \ldots, t_n) \) is a term of sort \( s \) for each operator \( f \in F_{s_1...s_n,s} \) and terms \( t_i \) of sort \( s_i \) for \( i \in \{1, 2, \ldots, n\} \).

The \( S \)-sorted set of \( \Sigma(X) \)-terms is denoted as \( T_\Sigma(X) \overset{\text{def}}{=} \{(T_\Sigma(X))_s\}_{s \in S} \). For the \( S \)-sorted empty sets of variables \( \{\} \), an \( \Sigma(\{\}) \)-term is called \( \Sigma \)-term or \( \Sigma \)-ground term. \( T_\Sigma \overset{\text{def}}{=} T_\Sigma(\{\}) \) denotes the \( S \)-sorted set of \( \Sigma \)-ground
terms. $T_\Sigma(X)$ and $T_\Sigma$ can be organized as $\Sigma$-algebras by using the above stated inductive definition of terms. $T_\Sigma$ as a $\Sigma$-algebra is the initial algebra of $\Sigma$-algebras as stated as follows.

**Fact 1.** [28, 29] For any $\Sigma$-algebras $A$ there exists a unique $\Sigma$-algebra morphism $T_\Sigma \to A$. □

The least sort $1s(t)$ of a term $t \in T_\Sigma(X)$ is defined to be the least sort $s$ such that $t \in (T_\Sigma(X))_s$. A regular signature $\Sigma$ [27,28] guarantees that each term $t \in T_\Sigma(X)$ has the unique minimum parse with the least sort (28,29).

### 2.2.4. Valuations and Term Interpretation

A valuation (substitution, instantiation) assigns values to variables, that is, instantiates each variable with a value in a given model. Let $\Sigma$ be $(S, \leq, F)$-signature. Given a $\Sigma$-model $A$ and an $S$-sorted set $X$ of variables, a valuation $\theta : X \to A$ is an $S$-sorted family of maps $\{\theta_s : X_s \to A_s\}_{s \in S}$. Each $\Sigma(X)$-term $t \in T_\Sigma(X)$ can be interpreted as a value $\theta(t)$ in the model $A$ as follows. That is, $\theta$ can extend to $\theta : T_\Sigma(X) \to A$.

- $\theta(t) = A_f$ if $t$ is a constant $f$.
- $\theta(t) = \theta(x)$ if $t$ is a variable $x$.
- $\theta(t) = A_f(\theta(t_1), \ldots, \theta(t_n))$ if $t$ is of the form $f(t_1, \ldots, t_n)$ for some $f \in F_{s_1 \ldots s_n}$ and terms $t_i \in (T_\Sigma(X))_{s_i}$ ($1 \leq i \leq n$).

Note that $\theta$ can be seen as a $\Sigma(X)$-algebra morphism $\theta : T_\Sigma(X) \to A$ by interpreting $\Sigma(X)$ as a signature with fresh constants $X$ (i.e., $\Sigma \cup X$).

### 2.2.5. $(S, \leq, F, F^c)$-Algebras

An $(S, \leq, F, F^c)$-algebra $A$ is an $(S, \leq, F)$-algebra with the carrier sets for the constrained sorts consisting of interpretations of terms formed with constructors and elements of loose sorts. That is, the following holds for $\Sigma^c = (S, \leq, F^c)$.

There exists an $S^c$-sorted set of loose variables $Y = \{Y_s\}_{s \in S^c}$ and a valuation $\theta : Y \to A = \{\theta_s : Y_s \to A_s\}_{s \in S^c}$ such that $\theta : T_{\Sigma^c}(Y) \to A$ is surjective for each constrained sort $s \in S^c$ (i.e., $\theta_s : (T_{\Sigma^c}(Y))_s \to A_s$ is a surjection for each $s \in S^c$).

**Example 2.** The CafeOBJ code

\[
\{[Elt<Seq] \ op \ \_\_ \ : \ Seq \ Seq \to Seq \ \{\text{constr assoc}\}.\}
\]
defines the constructor-based order-sorted signature

\[
\text{SEQsg} = ([Elt,\text{Seq}], \{\{\text{Elt,Seq}\}, \{\_\_\}, \{\_\_\}\}).
\]

Then $\text{NATsq}$ (sequences of natural numbers) defined by

- $\text{NATsq}_{\text{Elt}} = \{1, 2, \ldots\}$,
- $\text{NATsq}_{\text{seq}} = \{n_1 n_2 \cdots n_k \mid (1 \leq k), n_i \in \text{NATsq}_{\text{Elt}}(1 \leq i \leq k)\}$

is a $\text{SEQsg}$-algebra. It can be seen as follows. Let $Y = \{Y_{\text{Elt}}\}$, $Y_{\text{Elt}} = \{y_1, y_2, \ldots\}$ then

\[
(T_{\text{SEQsg}}(Y))_{\text{seq}} = \{z_1 z_2 \cdots z_k \mid (1 \leq k), z_i \in Y_{\text{Elt}}(1 \leq i \leq k)\}
\]

and $\theta_{\text{seq}} : (T_{\text{SEQsg}}(Y))_{\text{seq}} \to \text{NATsq}_{\text{seq}}$ is a surjection by taking $\theta_{\text{Elt}} : Y_{\text{Elt}} \to \text{NATsq}_{\text{Elt}}$, with $\theta_{\text{Elt}}(y_i) = i \ (1 \leq i)$. □
2.3. Sentences and Specifications

2.3.1. **Sigma-Equations**

A \( \Sigma \)-equation is a main sentence of CafeOBJ and is declared as follows; cq stands for conditional equation.

\[
\text{cq} \ l(X) = r(X) \text{ if } c(X)
\]

\( X \) is a finite \( \Sigma \)-variable set \( X = \{X_1, X_2, \ldots, X_n\} \), \( c(X) \in \Sigma(X) \) and \( l(X), r(X) \in T_\Sigma \), for some sort \( s \in S \). If \( c(X) \) is true (a built-in constant of the sort \( \text{Bool} \)) the equation is unconditional and written as “\( \text{eq} \ l(X) = r(X) \)”. Labels \( lb_1 \cdots lb_n \) can be put to an equation by declaring them just after the keyword \( \text{eq} \) or \( \text{cq} \) as “\( \text{eq}[lb_1 \cdots lb_n] : l(X) = r(X) \)”. A label is the builtin attribute \( \text{nonexec} \) or an ordinary label (see Example 3).

An equation is called **conjunctive** if \( c(X) \) does not contain the built-in equality predicate \( \_=_ \) or “\( (l_1(X) = r_1(X)) \) and \( \cdots \) and \( (l_n(X) = r_n(X)) \)” \( (1 \leq n) \) and each of \( l_i(X) \) and \( r_i(X) \) does not contain the equality predicate \( \_=_ \). An equation is called **sort decreasing** if \( \text{ls}(\theta(l(X))) \leq \text{ls}(\theta(r(X))) \) (see \( 2.2.3 \)) for \( \text{ls}(t) \) for any substitution \( \theta : X \rightarrow T_\Sigma \) such that \( \theta(x) = x' \) \( (x' \in X) \) and \( \text{ls}(x') \leq \text{ls}(x) \). A set of equations \( E \) is conjunctive or sort decreasing if each element of \( E \) is so. Conjunctiveness and sort decreasingness can be checked automatically, but not implemented in the current CafeOBJ system.

2.3.2. **Equational Specifications**

An **equational specification** \( SP \) is defined to be a pair of signature \( \Sigma \) and a set of \( \Sigma \)-equations \( E \), and denoted as \( SP = (\Sigma, E) \).

2.3.3. **Structured Specifications**

A specification \( SP \) can be defined by importing other already defined specification \( SP_a \) and declaring newly added signature \( \Sigma \) and equations \( E \), and denoted as \( SP = (SP_a; \Sigma, E) \). If \( SP_a = (\Sigma_a, E_a) \) then \( SP = (\Sigma_a \cup \Sigma, E_a \cup E) \).

It is a casual definition of structured specifications. More formal treatments can be found in [10, 18].

2.4. **Satisfaction**

2.4.1. **Equation Satisfaction** \( A \vDash_\Sigma e \)

An equation \( e = \text{"cq} \ l(X) = r(X) \text{ if } c(X) \text{"} \) is **satisfied** by a \( \Sigma \)-algebra \( A \), in symbols \( A \vDash_\Sigma e \), iff \( \theta(l(X)) = \theta(r(X)) \) whenever \( \theta(c(x)) = \text{true} \) for any valuation \( \theta : X \rightarrow A \). That is, an equation is satisfied by an algebra iff any possible way to assign values to variables evaluates both sides of the equation to the same value whenever the condition is evaluated to true. \( \Sigma \) of \( \vDash_\Sigma \) can be dropped if it is clear from the context. The following facts are implied from the fact that the built-in BOOL realizes the propositional calculus with equality.

**Fact 2.** Let \( \_=_ \) and \( \_\text{implies}_\_ \) be Boolean operators of the CafeOBJ’s built-in module BOOL, and \( (l(X) = r(X)), (c(X) \text{ implies } l(X) = r(X)) \in (T_\Sigma \text{Bool}) \).

(i) \( A \vDash_\Sigma \text{"eq} \ l(X) = r(X) \text{"} \Leftrightarrow A \vDash_\Sigma \text{eq}(l(X) = r(X)) = \text{true} \).

(ii) \( A \vDash_\Sigma \text{"cq} \ l(X) = r(X) \text{ if } c(X) \text{"} \Leftrightarrow A \vDash_\Sigma \text{eq}(c(X) \text{ implies } l(X) = r(X)) = \text{true} \). \( \Box \)
2.4.2. Mod(SP) and Loose/Tight Denotations

The set \( \text{Mod}(SP) \) of models of a specification \( SP = (\Sigma, E) \) is defined as \( \text{Mod}(SP) \equiv \{ A \in \text{Mod}(\Sigma) | \forall e \in E (A \models e) \} \). An element of \( \text{Mod}(SP) \) is called an \( SP\text{-algebra} \) (or \( SP\text{-model} \)).

A specification \( SP \) in CafeOBJ (i.e., a module) can have loose or tight denotation that are indicated by two keywords \text{mod}\* or \text{mod!} respectively (see CafeOBJ codes for the module TRIV and LIST in [3.1.02:-07]). The loose denotation intends to denote \( \text{Mod}(SP) \), and the tight denotation intends to denote the initial or free algebra of \( \text{Mod}(SP) \) assuming its existence. That is, tight denotation intends to specify a specific data type, and loose denotation intends to specify a class of data types with common constraints.

2.4.3. SP \models p(Y) and Theorem of Constants

Let \( SP = (\Sigma, E) \) be a specification, and \( Y \) be a \( \Sigma \)-variable set \( (2.3.5) \). A Boolean term \( p(Y) \) \( (T_{\Sigma, Y})_{\text{Bool}} \), which is supposed to express a property of interest for a specification \( SP \), is defined to be satisfied by \( SP \) (in symbols \( SP \models p(Y) \)) if \( \forall A \in \text{Mod}(SP) (A \models _{\Sigma} " \text{eq } p(Y) = \text{true}" ) \).

Let \( \Sigma \cup Y \) be a signature obtained by adding \( Y \) to \( \Sigma \) as fresh constants, and \( p^Y \in (T_{\Sigma \cup Y})_{\text{Bool}} \) be the term obtained from \( p(Y) \) by considering each \( y \in Y \) as a fresh constant. Let \( A \) be a \( \Sigma \)-algebra, \( \theta : Y \rightarrow A \) be a valuation, then there is a one to one correspondence between a pair \((A, \theta)\) and a \((\Sigma \cup Y)\)-algebra \( A' \). Hence the following Fact (Theorem 3.3.11 of [28]) which will be used to obtain Facts [12, 14] and play an important role in proof scores.

**Fact 3. (Theorem of Constants)** \((\Sigma, E) \models p(Y) \iff (\Sigma \cup Y, E) \models p^Y\)

The following facts are implied from the definition of \( \models \) and **Fact 3**

**Fact 4.** Let \( p, q \in (T_{\Sigma})_{\text{Bool}} \), \( p'(Y), q'(Y) \in (T_{\Sigma, Y})_{\text{Bool}} \), and \( p''(Y) \in (T_{\Sigma, Y})_{\text{Bool}} \) be the term gotten by substituting non overlapping strict subterms of \( p \) with variables in \( Y \).

\( p' \) and \( q' \) are obtained as follows:

\( p' = p(\theta_1, \ldots, \theta_n) \) where \( \theta_i \) are fresh variables.

\( q' = q(\theta_1, \ldots, \theta_n) \) where \( \theta_i \) are fresh variables.

\( p'' = p'(\theta_1, \ldots, \theta_n) \) where \( \theta_i \) are fresh variables.

\( q'' = q'(\theta_1, \ldots, \theta_n) \) where \( \theta_i \) are fresh variables.

2.5. Quotient Algebra \( T_{\Sigma, E} \) and Initial Algebras

A congruence \( \equiv \) on an \((S, \leq, F)\)-algebra \( A \) is an \( S \)-sorted equivalence on \( A \) (i.e., \( \{ s \leq A_s \times A_s | s \in S \} \) such that (i) if \( a_i \equiv a'_i \ (i \in \{1, \ldots, n\}) \)

---

3. Refers to the CafeOBJ code with line numbers from 02: to 07: in the Subsection 3.1.
then \( A_f(a_1, \ldots, a_n) \equiv_s A_f(a'_1, \ldots, a'_n) \) for each \( f \in F_{s_1 \ldots s_n} \), and (ii) if \( s \leq s' \) (\( s, s' \in S \)) and \( a, a' \in A_s \) then \( a \equiv_s a' \) if \( a \equiv_{s'} a' \).

For an equational specification \( SP = (\Sigma, E) \), \( \Sigma = (S, \leq, F) \), a quotient algebra \( T_{\Sigma, E} \) is constructed as follows:

- for each \( s \in S \) let \( (T_{\Sigma, E})_s \) be the set of equivalence classes of \( \Sigma \)-terms in \( (T_{\Sigma})_s \) under the congruence \( \equiv_E \) defined as \( (t \equiv_E t' \text{ iff } (\Sigma, E) \models t = t') \). That is, \( (T_{\Sigma, E})_s = \{ [t]_E \mid t \in (T_{\Sigma})_s \} \).
- each operator \( f \in F_{s_1 \ldots s_n} \) is interpreted as \( (T_{\Sigma, E})_t(f(t_1)_{\equiv_E}, \ldots, t_n)_{\equiv_E} = f(t_1, \ldots, t_n)_{\equiv_E} \) for all \( t_i \in (T_{\Sigma})_{s_i} \) (\( i \in \{1, \ldots, n\} \)) by using the congruence property of \( \equiv_E \) on \( T_{\Sigma} \).

2.5.1. The Initial Algebra of \( \mathbb{Mod}((S, \leq, F), E) \)

**Fact 5.** [28, 29] Let \( SP = (\Sigma, E) \), \( \Sigma = (S, \leq, F) \) with no constructors. If \( E \) is conjunctive, for any algebra \( A \in \mathbb{Mod}(SP) \) there exists a unique \( \Sigma \)-algebra morphism \( T_{\Sigma, E} \to A \). \( \square \)

2.5.2. The Initial Algebra of \( \mathbb{Mod}((S, \leq, F, F^c), E) \)

Let \( SP = ((S, \leq, F, F^c), E) \) be a constructor-based order-sorted specification and \( S^c \) be the set of constrained sorts and \( S^l \) be the set of loose sorts (2.1.6), and \( F^S^c \overset{\text{def}}{=} \{ f : w \to s \mid f \in F, s \in S^c \} \), and \( \Sigma^S^c \overset{\text{def}}{=} (S, \leq, F^S^c) \), and \( \Sigma^c \overset{\text{def}}{=} (S, \leq, F^c) \), and \( Y \) be any \( S^l \) sorted set of variables. A specification \( SP \) is defined to be **sufficiently complete** if for any term \( t \in T_{\Sigma, \Sigma^c}(Y) \) there exists a term \( t' \in T_{\Sigma, \Sigma^c}(Y) \) such that \( SP \models t = t' \).

**Fact 6.** [18, 24] Let \( SP = (\Sigma, E) \), \( \Sigma = (S, \leq, F, F^c) \) be a constructor-based order-sorted specification. If the specification \( SP \) is sufficiently complete and \( E \) is conjunctive, for any algebra \( A \in \mathbb{Mod}(SP) \) there exists a unique \( \Sigma \)-algebra morphism \( T_{\Sigma, E} \to A \). \( \square \)

2.6. Deduction, Reduction, CafeOBJ Execution

Let \( SP = (\Sigma, E) \), \( \Sigma = (S, \leq, F) \) be an equational specification. If there are operators in \( F \) that have associative and/or commutative attributes then let \( AC \) denote the set of associativity or commutativity (AC) equations (2.1.3) of the operators. Let \( =_{AC} \) denote the smallest \( \Sigma \)-congruence (2.5.1) on the \( \Sigma \)-term algebra \( T_{\Sigma} \) (2.2.3) that includes the equivalence relation defined by the equations in \( AC \). \( t =_{AC} t' \) can be decided with AC matching algorithm. If there are no AC operators in \( F \), \( AC \) is empty and \( =_{AC} \) is the usual equality = on \( T_{\Sigma} \).

Let \( \square \) be a special fresh constant of a sort \( s \in S \), and let \( t_c[\square_s] \) denote a ground term composed of operators in \( F \cup \{ \square \} \) with one \( \square \). The term \( t_c[\square_s] \) is called an \( F \cup \{ \square \} \)-term or a context, and \( t_c[t] \) denotes the \( \Sigma \)-term obtained by substituting \( \square \) with a term \( t \in (T_{\Sigma})_s \). Let \( p \in (T_{\Sigma}_e)_{\text{bool}} \) (i.e., a Boolean ground \( \Sigma \)-term).
2.6.1. Equational Deduction

An equational specification $SP = (\Sigma, E \cup AC)$ defines (i) **one-step deduction relation** $\leftrightarrow_E \subseteq T^*_\Sigma \times T^*_\Sigma$ and (ii) its reflexive and transitive closure $\leftrightarrow^*_E$ called **deduction relation**. Deduction (or inference) rules for $\leftrightarrow_E$ and $\leftrightarrow^*_E$ are given as follows for $t_l, t_r, t_m \in T^*_\Sigma$.

(a) If $(t_l =_{AC} t_r)$, $\Rightarrow$ $t_l \leftrightarrow^*_E t_r$. That is, $t_l \leftrightarrow^*_E t_r$ is deduced if $(t_l =_{AC} t_r)$.
(b) If $\neg(t_l =_{AC} t_r)$, $t_l \leftrightarrow^*_E t_m \land t_m \leftrightarrow^*_E t_r \Rightarrow t_l \leftrightarrow^*_E t_r$. That is, if $(t_l =_{AC} t_r)$ does not hold, $t_l \leftrightarrow^*_E t_r$ is deduced when $t_l \leftrightarrow^*_E t_m$ and $t_m \leftrightarrow^*_E t_r$ are deduced for some $t_m$; the following (c1), (c2) are interpreted in a similar way.
(c) For an equation "cq $l(X) = r(X) \text{ if } c(X)." \in E \cup AC$, a valuation $\theta : X \rightarrow T^*_\Sigma$, and a context $t_c[\square]$, 

\[
\begin{align*}
(c1) & \quad \theta(c(X)) \leftrightarrow^*_E \text{true} \Rightarrow t_c[\theta(l(X))] \leftrightarrow^*_{E/AC} t_c[\theta(r(X))], \\
(c2) & \quad \theta(c(X)) \leftrightarrow^*_E \text{true} \Rightarrow t_c[\theta(r(X))] \leftrightarrow^*_{E/AC} t_c[\theta(l(X))].
\end{align*}
\]

Let $SP \vdash p$ denote $p \leftrightarrow^*_E p$.

**Fact 7.** [28]

(i) $SP \vdash p \Rightarrow SP \models p$.
(ii) If $E$ is conjunctive, $SP \vdash p \iff SP \models p$. \hfill $\square$

Equational deduction can be done also on the equivalence classes defined by the congruence relation $=_E$, and an equational specification $SP = (\Sigma, E)$ defines **one-step deduction relation modulo AC** $\leftrightarrow^{E/AC}$ and **deduction relation modulo AC** $\leftrightarrow^{E/AC}_E \subseteq T^*_\Sigma \times T^*_\Sigma$. Deduction rules for $\leftrightarrow^{E/AC}_E$ and $\leftrightarrow^{E/AC}$ are given as follows for $t_l, t_r, t_m \in T^*_\Sigma$.

(a) If $(t_l =_{E/AC} t_r)$, $\Rightarrow$ $t_l \leftrightarrow^{E/AC}_E t_r$.
(b) If $\neg(t_l =_{E/AC} t_r)$, $t_l \leftrightarrow^{E/AC}_E t_m \land t_m \leftrightarrow^{E/AC}_E t_r \Rightarrow t_l \leftrightarrow^{E/AC}_E t_r$.
(c) For an equation "cq $l(X) = r(X) \text{ if } c(X)." \in E$, a valuation $\theta : X \rightarrow T^*_\Sigma$, and a context $t_c[\square]$, if $t_l =_{E/AC} t_c[\theta(l(X))] \land t_r =_{E/AC} t_c[\theta(r(X))]$, 

\[
\begin{align*}
(c1) & \quad \theta(c(X)) \leftrightarrow^{E/AC}_E \text{true} \Rightarrow t_l \leftrightarrow^{E/AC}_E t_r, \\
(c2) & \quad \theta(c(X)) \leftrightarrow^{E/AC}_E \text{true} \Rightarrow t_r \leftrightarrow^{E/AC}_E t_l.
\end{align*}
\]

The following is proved via a similar argument as Lemma 4 of [38].

(\textbf{E/AC}) \quad \forall t_l, t_r \in T^*_\Sigma (t_l \leftrightarrow^*_E t_r \iff t_l \leftrightarrow^{E/AC}_E t_r).

2.6.2. Rewriting Reduction

For a term $t \in T^*_\Sigma$ let $\text{var}(t) \subseteq X$ denote the set of variables in $t$. An equation "cq $[lbs] :: l(X) = r(X) \text{ if } c(X)."$ is called a **rewrite rule** if $\text{var}(r(X)) \cup \text{var}(c(X)) \subseteq \text{var}(l(X))$ and :\text{nonexec} is not in $lbs$. If $lbs$ is empty [$lbs = \emptyset$] : can be omitted. If each equation $e \in E$ is a rewrite rule, $SP$ is called a **reduction specification** (or **term rewriting system (TRS)**) and defines **one-step reduction relation modulo AC** $\rightarrow_{E/AC}$ and **reduction relation modulo AC** $\rightarrow^{*}_{E/AC} \subseteq T^*_\Sigma \times T^*_\Sigma$. Inference rules for $\rightarrow_{E/AC}$ and $\rightarrow^{*}_{E/AC}$ are the same as for $\leftrightarrow_{E/AC}$ and $\leftrightarrow^{*}_{E/AC}$ in [2.6.1] by replacing $\leftrightarrow_{E/AC}$ with $\rightarrow_{E/AC}$ and $\leftrightarrow^{*}_{E/AC}$ with $\rightarrow^{*}_{E/AC}$ and delete the rule (c2) because rewriting is only from left to right direction.

Let $SP \vdash p$ denote $p \rightarrow^{*}_{E/AC} \text{true}$. [2.6.1] (\textbf{E/AC}) implies the following.
Fact 8. \( SP \vdash p \Rightarrow SP \not\vdash p \) \qed

Example 3. An equation with the :nonexec attribute, which is not used for the application of the rule (c1), can be declared as

\[cq[\text{trans :nonexec}]: X = Z \text{ if } (X = Y \text{ and } Y = Z)\]

where \(X, Y, Z\) are variables of some sort. This equation is logically valid because of transitivity of equality, but is not a rewrite rule because \(\var{Z} \cup \var{(X = Y \text{ and } Y = Z)} \subseteq \var{X}\) does not hold. This equation can become an executable rewrite rule by instantiating its variables with ground terms, and deleting the :nonexec attribute, as the equation \(\{\text{true}\}\) is instantiated with the :init command declared at \(\{1.2.3\}^{11:13}\) and applied at \(\{1.2.3\}^{21:22}\). \qed

2.6.3. CafeOBJ Execution

Let \(SP = (\Sigma, E)\) be a reduction specification (or TRS), then \(SP\) defines one-step weak reduction relation modulo AC and weak reduction relation modulo AC \(\rightarrow_{\Sigma, AC}, \rightarrow^*_\Sigma, AC \subseteq T_\Sigma \times T_\Sigma\). Inference rules for \(\rightarrow_{\Sigma, AC}, \rightarrow^*_\Sigma, AC\) are given as follows for \(t_l, t_r, t_m, t_p \in T_\Sigma\).

(a) If \((t_l \equiv_{AC} t_r) \Rightarrow t_l \rightarrow^*_\Sigma, AC t_r\).
(b) If \((t_l \equiv_{AC} t_r), t_l \rightarrow^*_\Sigma, AC t_m \land t_m \rightarrow^*_\Sigma, AC t_r \Rightarrow t_l \rightarrow^*_\Sigma, AC t_r\).

c1) For a rewrite rule \(\text{cq}(l(X) = r(X) \text{ if } c(X)), \in E\), a valuation \(\theta : X \rightarrow T_\Sigma\), and a context \(\text{t}_c[\text{\[]}_\theta\text{\]}\), if \(t_p \equiv_{AC} \theta(l(X))\),

\(\theta(c(X))) \rightarrow^*_\Sigma, AC \text{ true} \Rightarrow t_c[t_p] \rightarrow_{\Sigma, AC} t_c[\theta(r(X))].\)

The check with \(\equiv_{AC}\) in (c1) via AC-matching is restricted to \(t_p \equiv_{AC} \theta(l(X))\), and \(\rightarrow_{\Sigma, AC}, \rightarrow^*_\Sigma, AC\) can be implemented much more efficiently than \(\rightarrow_{\Sigma, AC}, \rightarrow^*_\Sigma, AC\).

(a) and (b) tell that \(t_0 \rightarrow^*_{\Sigma, AC} t_n\) if there exists a sequence of one-step reductions \(t_0 \rightarrow_{\Sigma, AC} t_1 \cdots t_{n-1} \rightarrow_{\Sigma, AC} t_n\) or \(t_0 \equiv_{AC} t_n\).

(c1) tells the way to compute \(t_r\) from \(t_l\) for establishing a one-step reduction \(t_l \rightarrow_{\Sigma, AC} t_r\) if a context \(t_c[\text{\[]}_\theta\text{\]}\) and a rewrite rule \(\text{cq}(l(X) = r(X) \text{ if } c(X)), \in E\) and are chosen. If \(b = \theta(c(X))\) then (c1) declares that \(b \rightarrow^*_\Sigma, AC \text{ true}\) is a prerequisite for establishing \(t_l \rightarrow_{\Sigma, AC} t_r\). Let \(\{b\}\) denote the necessary computation for checking \(b \rightarrow^*_\Sigma, AC \text{ true}\), then the computation step for establishing \(t_l \rightarrow_{\Sigma, AC} t_r\) is written as \(t_l \{b\} t_r\) and called a one-step conditional reduction (oc-red for short).

For a Boolean ground term \(b_0 \in (T_\Sigma)_{\text{Bool}}\), let \(\{b_0\}\) denote a sequence of one-step reductions from \(b_0\) to \text{true}. That is, \(\text{let } \{b_0\} = \text{"b_0 \rightarrow_{\Sigma, AC} b_1 \cdots b_n \rightarrow_{\Sigma, AC} \text{true}"} \text{ (0 \leq n) if } \neg(b_0 = \text{true})\), and \(\{b_0\} = \text{true}\) if \(b_0 = \text{true}\). \(\{b_0\}\) is defined as follows based on \(\{b_0\}\), where \(b_{i+1}\) \((0 \leq i \leq n)\) is the condition (i.e., \(\theta(c(X))\)) of the one-step reduction \(b_i \rightarrow_{\Sigma, AC} b_{i+1}\).

\[
\begin{align*}
\{b_0\} &= \{b_{01}\} b_1 \cdots b_i \{b_{i+1}\} b_{i+1} \cdots b_n \{b_{n+1}\} \text{ true} \\
\text{if } \{b_0\} &= \text{false} \Rightarrow \rightarrow_{\Sigma, AC} b_1 \cdots b_i \rightarrow_{\Sigma, AC} b_{i+1} \cdots b_n \rightarrow_{\Sigma, AC} \text{ true}, \text{ and} \\
\{b_0\} &= \{\text{false}\} \text{ if } \{b_0\} = \text{true}.
\end{align*}
\]

\(\{\_\}\) applies recursively and can be nested infinitely as shown in Example 5 below, and \(\{b_0\}\) might be undefined because (i) it continues infinitely or (ii) it stops in the middle without getting to \text{true}. The above definition implies that
$t_0\{b_0\} t_1 \cdots t_i \{b_i\} t_{i+1} \cdots t_n \{b_n\} t_{n+1}$ \((0 \leq n)\) is a sequence of oc-reds iff each \(b_i\) of \(\{b_i\}\) \((0 \leq i \leq n)\) is a sequence of oc-reds or true.

A term \(t \in T_S\) is called \(\rightarrow_{E,AC}\)-reduced modulo AC, or simply just reduced, if there is no term \(t' \in T_S\) such that \(t \rightarrow_{E,AC} t'\). For a ground term \(t_0 \in T_S\), a CafeOBJ’s reduction command “reduce in SP: \(t_0\)” computes a sequence of oc-reds \(t_0 \{b_0\} t_1 \cdots t_i \{b_i\} t_{i+1} \cdots t_n \{b_n\} t_{n+1}\) from left to right as much as possible by choosing a context and a rewrite rule for each oc-red using a predefined algorithm, and, if the sequence terminates, returns the reduced term \(t_{0\text{red}}\) that ends the sequence. Note that \(t_{0\text{red}}\) is uniquely determined if the reduction terminates, and \(\_\text{red}\) is a partial function.

**Example 4.** The following CafeOBJ code declares, with the keyword mod!, a module (i.e., a specification) \(\text{PNAT+}\) (Peano NATural numbers with \(\_+\_\)). \(\text{PNAT+}\) declares between \(\{\}\) (i) three sorts \(\text{Zero}, \text{NzNat}, \text{Nat}\) with subsort relations, (ii) two constructor operators \(0, s_{\_}\) and one non-constructor operator \(\_+\_\), (iii) two variables \(X, Y\) after the keywords \(\text{vars}\), and (iv) two equations that reduce a term with non-constructors to a term only with constructors.

```cafeobj
mod! \text{PNAT+} \{ 
  [\text{Zero NzNat < Nat}] 
  \text{op} 0 : \rightarrow \text{Zero} \{\text{const}\} . 
  \text{op} s_{\_} : \text{Nat} \rightarrow \text{NzNat} \{\text{const}\} . 
  \text{op} \_+\_ : \text{Nat Nat} \rightarrow \text{Nat} . 
  \text{vars} \ X \ Y : \text{Nat} . 
  \text{cq} X + Y = Y \text{if} (X = 0) . 
  \text{eq} (s \ X) + Y = s(X + Y) . 
\}
```

The CafeOBJ command: “red in PNAT+: \(s 0 + 0\)” returns \(s 0\) after computing the following sequence of oc-reds.

\(s(0) + 0 \{\text{true}\} s(0 + 0) \{0 = 0 \{\text{true}\} \text{true}\} s 0 \) □

Since \(\rightarrow_{E,AC} \subseteq \rightarrow_{E/AC}\), it could happen that \((t \rightarrow_{E/AC} t')\) but \(! (t \rightarrow_{E,AC} t')\). CafeOBJ’s implementation of \(\rightarrow_{E,AC} \subseteq \rightarrow_{E/AC}\) is complete enough such that \((E,AC)\) \(\forall t, u \in T_S(t \rightarrow_{E,AC} u \Rightarrow \exists u' \in T_S(t \rightarrow_{E/AC} u' \wedge u =_{AC} u'))\).

This is realized by extending (or completing) each equation \(e \in E\) involving operators with AC attributes.

A TRS \(SP = (\Sigma, E)\) is defined to be

(i) **operationally terminating** \([34]\) if there is no infinite sequence of oc-reds or infinitely nested oc-reds no matter what context and rewrite rule are chosen for each oc-red,

---

4 All the CafeOBJ modules and proof scores explained in this paper are posted at the following web page:

https://cafeobj.org/~futatsugi/misc/apsco-220907/

and reader can execute them. The module \(\text{PNAT+}\) is in the file \text{peano-nat-spc.cafe} on the Web.

5 [38] provides the most advanced study of conditional order-sorted rewriting modulo axioms including important issues like AC-extended equations and \(\rightarrow_{E/AC}\) vs. \(\rightarrow_{E,AC}\).

6 [34] defines operational termination based on applications of inference rules and does not introduce oc-red.
(ii) **terminating** iff $\rightarrow_{E,AC}$ is well-founded,

(iii) **confluent** iff

\[ \forall t_1, t_2, t_3 \in T_{\Sigma}((t_1 \rightarrow_{E,AC} t_2 \land t_1 \rightarrow_{E,AC} t_3) \Rightarrow \exists t_4 \in T_{\Sigma}(t_2 \rightarrow_{E,AC} t_4 \land t_3 \rightarrow_{E,AC} t_4)). \]

(iv) **sufficiently complete** iff $t \rightarrow_{E,AC}^* t'$ for $t$ and $t'$ in $\Sigma$ and $Y$ being considered to be the set of fresh constants.

All the above four properties (i), (ii), (iii), (iv) are undecidable, but usable sufficient conditions for guaranteeing them are known.

**Example 5.** If a TRS has only one equation “cq b = true if b .” for a fresh Boolean constant $b$, the system is terminating because (i) the builtin module BOOL without the equation is terminating, and (ii) “$b \rightarrow_{E,AC} t$” does not hold for any $t \in T_{\Sigma}$. The TRS is, however, not operationally terminating because there is an infinitely nested oc-reds “b {b {b · · ·}”.

Let $SP \models p$ denote that “reduce in $SP: p$.” returns true, i.e., $p^{\text{red}} = \text{true}$. The following is obtained via **Facts** 7, 8.

**Fact 9.** (PR1) 

$SP \models p \Rightarrow SP \models p$

Note that the above Facts hold even if $SP$ has no initial model, or is not operationally terminating, terminating, confluent, or sufficiently complete.

If a reduction specification (TRS) $SP = (\Sigma, E)$ is (1) conjunctive, i.e., $E$ is conjunctive (2.3.1), (2) sort decreasing, i.e., $E$ is sort decreasing (2.3.1), (3) operationally terminating, (4) confluent, and (5) sufficiently complete then $SP$ is called **red-complete**.

**Fact 10.** \[38\] For a red-complete reduction specification $SP = (\Sigma, E)$,

\[ \forall t, u \in T_{\Sigma}(t =_{E \cup AC} u) \Leftrightarrow (t^{\text{red}} =_{AC} u^{\text{red}}) \].

3. Basic Proof Scores

Basic techniques for constructing specifications and proof scores are presented, and justified by the Facts shown in Section 2.

3.1. Specifying Systems

The following CafeOBJ code 01:-12: specifies a generic list data structure. Numbers in 1st to 3rd columns like 01:, 10: are for explanation, and CafeOBJ code starts from the 7th column. CafeOBJ code starting -- or --> followed by a blank, like in 01: or 08:, is a comment until the end of a line. A comment with “--” or “-->” can start at any point of a line.

---

7Refers to the CafeOBJ code with line numbers from 01: to 12: in this subsection. See also the code reference notation explained in the footnote 8.

8A footnote on # shows in which files the following CafeOBJ code exists. 01:-17: is in the file list-append.cafe on the Web: [https://cafeobj.org/~futatsugi/misc/apsco-220907/](https://cafeobj.org/~futatsugi/misc/apsco-220907/).
-- generic collection of objects
mod* TRIV {{Elt}} -- TRIV is a builtin module

-- generic list
mod! LIST (X :: TRIV) {
  [NnList < List]
  op nil : -> List {constr}
  op _|_ : Elt List -> NnList {constr}
}

-- generic list with enhanced _=_
mod! LIST= (X :: TRIV) {
  pr(LIST(X))
  eq (nil = E:Elt | L:List) = false.
  eq (E1:Elt | L1:List = E2:Elt | L2:List) = (E1 = E1)and(L1 = L2).
}

The keyword mod* (02:) declares, with loose denotation (2.4.2), a module
with name TRIV and its body {{Elt}} that just declares the sort
Elt
with no constraints. The keyword mod! (04:) declares, with tight denotation
(2.4.2), a module with name LIST, its parameter list (...) (04:), and its body
{} (04:-07:). The parameter list (X :: TRIV) declares that this module
has a parameter module X, and that can be replaced by a module that satisfies
the module TRIV, i.e., by any module with at least one sort. The body declares,
i) two sorts NnList and List with the former being a subsort (2.1.1) of the
latter (05:), (ii) constructor operator (2.1.0) nil with rank (2.1.0) List (06:)
and constructor operator _|_ with rank “Elt List NnList” (07:).

The module LIST= declares two equations (11:-12:) for enhancing the
builtin equality _=_ on the sort List. E:Elt, L1:List, etc. are on-line variable
declarations each of them is effective until the end of each equation. pr(LIST(X))
(10:) declares that the module (LIST(X)) is imported in protecting mode, i.e.,
with no changes on its models.

Because the module LIST has the tight denotation and its model is the
initial algebra of Mod(LIST) (2.4.3), the equality of the initial model does not
change with the addition of the two equations. It is sometimes necessary to add
equations to refine the definition of the equality on the initial algebra, for only
one equation: “eq (L:List = L) = true .” is builtin originally.

The following module APPEND (14:-17:) specifies the append operator _#_
of rank “List List List” via the two equations (16:-17:).

13:  --> append operator _#_ on List
14:  mod! APPEND (X :: TRIV, Y :: LIST(X)) {
15:    op _#_ : List List -> List.
16:    eq nil # L2:List = L2.
17:    eq (E:Elt | L1:List) # L2:List = E | (L1 # L2).
}

The parameter declaration (X :: TRIV, Y :: LIST(X)) (01:) includes the 2nd
parameter “Y :: LIST(X)” and also indicates that the module LIST(X) is
imported in protecting mode. The 2nd parameter is prepared for instantiating it
with a more elaborated module LIST=(). That is, we need
APPEND(X, LIST=() instead of APPEND(X, LIST())
in some occasion (see Y.4.10:).
3.2. Specifying Properties of Interest

Assume that we want to construct a proof score for proving that the operator \( \# \) of the module APPEND is associative. For expressing the proof goal, the following module APPENDassoc (01:) defines the predicate appendAssoc and the module APPENDassoc@ (06:) introducing the three fresh constants \( l1@, l2@, l3@ \) of the sort List (08:).

01: mod APPENDassoc (X :: TRIV,Y :: LIST(X)) {  
02: ex(APPEND(X,Y))  
03: pred appendAssoc : List List List .  
04: eq appendAssoc(L1:List,L2:List,L3:List) =  
05: ((L1 # L2) # L3 = L1 # (L2 # L3)) . }  

06: mod APPENDassoc@ (X :: TRIV,Y :: LIST(X)) {  
07: ex(APPENDassoc(X,Y))  
08: ops l1@ l2@ l3@ : -> List . }  
09: mod APPENDassoc@= (X :: TRIV) {  
10: pr(APPENDassoc@=(X,LIST=(X))) }  

The keyword mod (01:,06:,09:) indicates no specific intention of the loose or tight denotation of the module, and used for proof modules in proof scores. The declaration ex(...) (02:,07:) indicates the importation of a module in extending mode. That is, no change except the addition of new operators like \( l1@, l2@, l3@, \) appendAssoc. The module APPENDassoc@= (09:) is using the module LIST= instead of LIST by putting LIST= to the 2nd parameter of APPENDassoc@ (10:), then of APPENDassoc@= (07:), and then of APPEND (02:).

The associativity of the operator \( \# \) is formalized as

APPENDassoc@= ⊨ appendAssoc(l1@,l2@,l3@)  
and by Fact 9

APPENDassoc@= ⊢ appendAssoc(l1@,l2@,l3@)  
is sufficient for the proof. However, the CafeOBJ reduction:

red in APPENDassoc@= : appendAssoc(l1@,l2@,l3@)  
does not return true, where red is the abbreviation of reduce.

3.3. Case-Split and Induction

Let \( M = (\Sigma, E) (\Sigma = (S, \leq, F, F^c)) (2.1.0) \) be a module (i.e., a specification) in CafeOBJ. If \( M \vdash p \) does not hold, the universal strategy for proving \( M \vdash p \) is making use of a lemma (including for discharging a contradictory case), a case-split, and/or an induction. Specification \( M \) may need some improvements as well.

3.3.1. Case-Split with Exhaustive Equations

Let \( Y_i = \{ Y_{i_s} \}_{s \in S} \) be an \( S \)-sorted set of fresh constants and \( e_i \) be a \((\Sigma \cup Y_i)\)-equation (2.3.4). That is, \( e_i \) may contain fresh constants in \( Y_i \) (i.e., in \( Y_{i_s} \) for

\[01:-10: \text{is in the file list-append.cafe on the Web.}\]
some $s \in S$ that are not in $\Sigma$. An $M$-model $A \in \text{Mod}(M)$ \[2.4.2\] is defined to satisfy ($\Sigma \cup T$)-equation $e_i$ (in symbols $A \models_{\Sigma \cup T} e_i$) iff $A \models_{\Sigma} e_i$ \[2.4.1\] holds for some interpretation of every element in $Y_i$ in $A$ (i.e., when for each $s \in S$ every constant in $Y_i$ is interpreted as some element in $A_s$). Equations $e_1, \cdots, e_n$ (1 $\leq$ $n$) are defined to be exhaustive for $M$ iff $\forall A \in \text{Mod}(M) (\exists i \in \{1, \cdots, n\} (A \models_{\Sigma \cup Y_i} e_i))$.

Let $e_1, \cdots, e_n$ (1 $\leq$ $n$) be exhaustive equations and $M_{+e} = (\Sigma \cup Y_i, E \cup \{e_i\})$ (1 $\leq$ $i$ $\leq$ $n$), then each $M$-model $A \in \text{Mod}(M)$ is an $M_{+e}$-model $A \in \text{Mod}(M_{+e})$ for some $i \in \{1, \cdots, n\}$ by interpreting $Y_i$ appropriately in $A$, and $A \models_{\Sigma} p$ if $M_{+e} \models p$. Hence the following proof rule on which every case-split in CafeOBJ can be based.

**Fact 11. (Case-Split with Exhaustive Equations)**

(PR2) \((M_{+e_1} \models p \land M_{+e_2} \models p \land \cdot \cdot \cdot \land M_{+e_n} \models p) \Rightarrow M \models p\) \(\square\)

Because the sort List of the module LIST \([5.4.04:-07:]\) is constrained \([2.4.6]\) with the two constructors nil and _L_, two equations “eq[e1]: $1\text{lo} = \text{nil} \_\_”, “eq[e2]: $1\text{lo} = \text{e}\$_1 \cup 1\text{l}\$_1” with fresh constants $\text{e}\$_1, 1\text{l}\$_1$ are exhaustive for the module APPENDassoc@ (see \([2.2.5]\) for the models of APPENDassoc@).

Let $M' = \text{APPENDassoc@}$ and $p' = \text{APPENDassoc}(1\text{l}\$_1, 1\text{d}\$_1, 1\text{r}\$_1)$ then \((M_{+e_1} \models p' \land M_{+e_2} \models p') \Rightarrow M' \models p'\) holds by (PR2), and the following CafeOBJ code \(01:-09:\) can serve to check whether $M_{+e_1} \models p' \land M_{+e_2} \models p'$ holds, \(01:-04:\) for $+_e1$ and \(05:-09:\) for $+_e2$.

\begin{verbatim}
01:    open APPENDassoc@ .
02:    eq 1l@ = nil .
03:    red appendAssoc(1l1@,1d1@,1r1@) .
04:    close
05:    open APPENDassoc@ .
06:    op e$ : -> Elt . op l1$ : -> List .
07:    eq 1l@ = e$ | 1l$. 
08:    red appendAssoc(1l1@,1d1@,1r1@) .
09:    close
\end{verbatim}

The “open $M$” command creates a new tentative module where all the $M$’s contents are imported, new sorts/operators/equations can be declared, and reduce commands can be executed. red is a shorthand for reduce, and if $M$ is opened, “red $p$.” stands for “red in $M$+ : $p$.” where the module $M$+ includes $M$’s contents plus the contents declared in the opened tentative module before the red command. The command close deletes the created module.

If the red commands at 03: and 08: would return true, $M_{+e_1} \models c_p' \land M_{+e_2} \models c_p'$ is proved, and by Fact 9 (PR1) $M' \models p'$ is proved. 03: returns true but 08: does not, and induction should be necessary.

### 3.3.2. Structural Induction

The standard way of induction for algebraic specifications is structural induction \([\square]\) on initial algebras (e.g., Theorem 6.5.9 and its Corollary of \([28]\)).

---

\(^{10}\)01:-09: is in the file list-append.cafe on the Web.
which can apply to any constructor algebra (i.e., an \((S, \le, F, F')\)-algebra of
and formulated as follows.

Let \(\Sigma = (S, \le, F, F')\), and \(p(y_1, \ldots, y_n) \in T_\Sigma(Y) (1 \le n)\) be a Boolean term with
finite variables \(Y = \{y_1, \ldots, y_n\}\) that describes a property of interest for a
module (specification) \(M\). The induction should be applied to a variable \(y_i \in Y\)
of a constrained sort \((2.7.0)\), and assume that the sort of \(y_1\) is \(\hat{s}\) and \(\hat{s}\) is a
 constrained sort without loss of generality. Let \(G^s \subseteq F^s\) be the set of constructors
\((2.7.0)\) for the sort \(\hat{s}\), i.e., \(G^s\) is the sorted sets \(G^s = \{G_w | w \in S^s, s \in S, s \le \hat{s}\}\).
The principle of structural induction is formulated as follows, where \(c_i\) is a fresh
constant of the sort \(s_i (1 \le i \le m)\).

\[
\left[ \forall G_1, \ldots, G_m, s \in G^\hat{s} \ (0 \le m) \right] \\
\left( \forall y \in G_1, \ldots, G_m, s \right) \\
\left[ \left( \forall (1 \le i \le m) \land (s_i \le \hat{s}) \right) \left( M \cup \{c_1, \ldots, c_m\} \models p(c_1, y_2, \ldots, y_n) \right) \right] \\
\Rightarrow M \models p(y_1, \ldots, y_n)
\]

By \((p_1 \land \cdots \land p_1) \Rightarrow q) \iff (p_1 \Rightarrow (\cdots (p_1 \Rightarrow q) \cdots))\) and \(\text{Fact 4(iv)}\),
\[
\left[ \left( \forall (1 \le i \le m) \land (s_i \le \hat{s}) \right) \cdots \right] \left( \models \right.
\]
the premise can be rewritten as follows. Note
that \(\{eq \cdot \cdots \cdot | (1 \le i \le m) \land \cdots \} \) is empty if \(m = 0\).

\[
\left[ \left( \forall (1 \le i \le m) \land (s_i \le \hat{s}) \right) \cdots \right] \left( \models \right.
\]
\[
M \models p(c_1, \ldots, c_m) \models y_1, \ldots, y_n
\]

\(\hat{\text{Fact 12. (Structural Induction)}}
\]
\[
\left[ \forall G_1, \ldots, G_m, s \in G^\hat{s} \ (0 \le m) \right] \\
\left( \forall y \in G_1, \ldots, G_m, s \right) \\
\left[ \left( \forall (1 \le i \le m) \land (s_i \le \hat{s}) \right) \left( M \cup \{c_1, \ldots, c_m\} \models \text{eq} p(c_1, y_2, \ldots, y_n) = \text{true} . \ | (1 \le i \le m) \land (s_i \le \hat{s}) \right) \right] \\
\Rightarrow M \models p(y_1, \ldots, y_n)
\]

\[\text{\text{Fact 13. (Induction on } s)}\]

\(01:-04: \) is a proof score for the premise’s conjunct \(\forall y \in G^s (\cdots)\)
(i.e., for \(m = 0\)), and the following \(01:-07: \) is a proof score for the premise’s
conjunct \(\forall y \in G_{s_1 \ldots s_m} (\cdots)\) (i.e., for \(m > 0\)) of \(\text{Fact 12}\) with the following
 correspondences:

\(\hat{s} \Rightarrow \text{List}, G^\hat{s} = \{\text{nil}, \{\_\}_\}, G_s = \{\text{nil}\}, G_{s_1 \ldots s_m} = \{\_\}_\),
\(p \Rightarrow \text{appendassoc}, Y^\alpha = \{110, 120, 130\}, M \cup Y^\alpha = \text{APPENDassoc}_\alpha =
\{c_1, \ldots, c_m\} = \{\text{e}$\text{s}, \text{i}$\text{1}$\}, \{y_1, \ldots, y_n\} = \{\text{L1, L2, L3}\}.

\[\text{\text{APPENDassoc}_\alpha = .}\]

\[01: \text{ open APPENDassoc}_\alpha .\]

\[01:-07: \text{ is in the file list-append.cafe on the Web.}\]
The left hand side of the equation 04:-05: is the reduced form of `appendAssoc(l1$,L2:List,L3:List)`, because left hand sides should be reduced for being effective as reduction rules. The `red` command 06: returns `true` and the proof score (3.3.1 + 01:-04: + 01:-07:) is effective for proving `APPENDassoc@= ⊨ appendAssoc(l1$,120$,130`) .

A proof score is called effective if the score constructs an effective proof tree (see 4.1).

**Example 6. (_=_ on List)**

Suppose that the equality `_=_ on the sort List is not enhanced like 3.7: 08:-12: and `APPENDassoc@` is used instead of `APPENDassoc@` in 01: above, then “`red appendAssoc(110$,120$,130`)” (06:) returns `(e$ | ((11$ # 120$) # 130$)) = (e$ | (11$ # (120$ # 130$)))` instead of `true`. This clearly shows that axioms 7.11:-12: are necessary to get the effective proof score. It is a nice example of a reduction result telling necessary axioms. This example also shows how module restructuring is inspired by observing reduction results thanks to powerful and flexible CafeOBJ’s module system.

### 4. Advanced Proof Scores

In advanced proof scores, case-split is realized through commands of the proof tree calculus (PTcalc), and induction is realized through well-founded induction (WFI) on argument tuples of a goal property predicate. As a result, advanced proof scores are (i) succinct and transparent for case-split, (ii) support a variety of induction schemes including multi-arguments induction.

#### 4.1. Proof Tree Calculus (PTcalc)

PTcalc is a refined version of a CafeOBJ version of CITP [27], and helps to prove `M ⊨ p` for a module `M = (Σ, E)`.

We already have the following proof rule (Fact 9).

(PR1) \( M ⊨ p \Rightarrow M ⊨ p \)

Usually `M ⊨ p` is difficult to prove directly, and we need to find exhaustive equations \( e_1, \ldots, e_n \) and make use of the following proof rule of case-split (Fact 11) with the exhaustive equations.

(PR2) \( (M_{+e_1} ⊨ p \land M_{+e_2} ⊨ p \land \cdots \land M_{+e_n} ⊨ p) \Rightarrow M ⊨ p \)

\( M_{+e_i} \vdash p \) would be still difficult to prove and (PR2) is applied repeatedly. The repeated applications of (PR2) generate proof trees successively. Each of the generated proof trees has the root node `M ⊨ p` and each of other nodes
is of the form \( M_{+e_{i_1} \ldots +e_{i_m}} \vDash p \) (1 \( \leq m \)) that is generated as a **child node** of \( M_{+e_{i_1} \ldots +e_{i_{m-1}}} \vDash p \) by applying (PR2). A **leaf node** (i.e., a node without child nodes) \( M_{+e_{i_1} \ldots +e_{i_k}} \vDash p \) (0 \( \leq k \)) of a proof tree is called **effective** if \( M_{+e_{i_1} \ldots +e_{i_k}} \vDash p \) holds. A proof tree is called effective if all of whose leaf nodes are effective. PTcalc proves \( M \vDash p \) by constructing an effective proof tree whose root node is \( M \vDash p \).

### 4.1.1. PTcalc Commands

PTcalc consists of the following commands with the keywords with the head character : that distinguishes them from ordinary CafeOBJ commands: 

- **:goal** declares goal propositions to be proved; 
- **:def** defines a new command; 
- **:apply** applies defined/pre-defined commands; 
- **:csp** declares a case-split command; 
- **:init** declares an initialize command; 
- **:red** does a reduction at the current goal; 
- **:show** shows info on proof; 
- **:describe** shows detailed info on proof; 
- **:select** selects the goal (node) specified; 
- **:set** sets parameters of PTcalc.

In PTcalc, a node in a proof tree is a special CafeOBJ module and called a **goal**. Each goal \( gl \) has a **name** like \( root \) or 1-2-3 (i.e., 3rd child of 2nd child of 1st child of \( root \)), and consists of the following five items.

1. The **next target (or default)** goal Boolean tag \( NTG(gl) \) that indicates a goal where a PTcalc command is executed. \( NTG(gl) = true \) holds for at most one goal in a proof tree.
2. The **context module** \( CTM(gl) \) that is a CafeOBJ module and corresponds to \( M \) of \( M \vDash p \). The goal \( gl \) inherits (imports) all the contents of \( CTM(gl) \).
3. The set of **introduced axioms (assumptions)** \( INA(gl) \) that corresponds to \( +e_{i_1} \ldots +e_{i_m} \) of \( M_{+e_{i_1} \ldots +e_{i_m}} \vDash p \), i.e., \( INA(gl) = \{ e_{i_1}, \ldots, e_{i_m} \} \).
4. The set of sentences (or equations) to be proved \( STP(gl) \) that corresponds to \( p \) of \( M_{+e_{i_1} \ldots +e_{i_m}} \vDash p \).
5. The **discharged** Boolean tag \( DCD(gl) \) that indicates whether \( gl \) is already discharged (i.e., proved).

\( (CTM(gl) \cup INA(gl)) \vDash STP(gl) \) corresponds to \( M_{+e_{i_1} \ldots +e_{i_m}} \vDash p \), where \( (CTM(gl) \cup INA(gl)) \) is understood as the module obtained by adding all the equations in \( INA(gl) \) to \( CTM(gl) \), and \( STP(gl) \) is understood as the conjunction of its elements. \( gl \) sometimes means \( (CTM(gl) \cup INA(gl)) \).

\[ \text{(47)} \]
defines the PTcalc commands by specifying in which way the above five items are changed by each command. For example, the case-split command **:csp** is defined as follows.

Let (a) \( tg \) be a goal such that \( NTG(tg) = true \) and (b) \( csid \) be the name of a **:csp** command defined by "**:def** \( csid = \text{:csp}\{ eq_1 eq_2 \cdots eq_n \} \)" with \( n \in \{1, 2, \cdots, \} \) equations \( eq_i \) (\( i \in \{1, 2, \cdots, n\} \)). Then executing the command "**:apply**(csid)" generates \( n \) sub-goals (child goals) \( tg-1, tg-2, \cdots, tg-n \) of \( tg \) as follows.

1. **(1:1)** Change \( NTG(tg) \) from true to false.
2. **(1:2)** \( NTG(tg-1) = true \).
3. **(1:3)** \( NTG(tg-i) = false \) (\( i \in \{2, \cdots, n\} \)).
(2) \(\text{CTM}(tg-i) = \text{CTM}(tg)\) \((i \in \{1,2,\cdots,n\})\).
(3) \(\text{INA}(tg-i) = \text{INA}(tg)\cup \{eq_i\} \ ((i \in \{1,2,\cdots,n\})\).
(4) \(\text{STP}(tg-i) = \text{STP}(tg)\) \((i \in \{1,2,\cdots,n\})\).
(5) \(\text{DCD}(tg-i) = \text{false} \ ((i \in \{1,2,\cdots,n\})\).

4.1.2. Proof Scores with PTcalc

The following proof score 01:-12: with PTcalc commands 05:-12: corresponds to the open...close style proof score (3.3.2 01:-04:) + (3.3.2 01:-07:) and proves \(\text{APPENDassoc}@\equiv \text{appendAssoc}(110,120,130)\).

01: \mod \text{APPENDassocPtc}(X :: \text{TRIV}) \{ 02: \op e\$ : \rightarrow \text{Elt} . \op l1\$ : \rightarrow \text{List} . \}
03: \select \text{APPENDassocPtc} .
04: \:goal \{eq \text{appendAssoc}(110,120,130) = \text{true} .\}
05: \:def l1\@ = :\text{csp}\{eq[\text{e1}]: l1\@ = \text{n1} \}
06: \:eq[\text{e2}]: l1\@ = e\$ | l1\$ .\}
07: \:apply(l1\@ \text{rd-})
08: \def iHyp =
10: \:init (eq \text{appendAssoc}(L1:\text{List},L2:\text{List},L3:\text{List}) = \text{true} .)
11: \by \{L1:\text{List} \leftarrow l1\$;\}
12: \:apply(iHyp \text{rd-})

03: corresponds to X.X.X.02: and the fresh constants e\$ and l1\$ are added to the module \(\text{APPENDassoc}@\equiv (02:)\) for defining the module \(\text{APPENDassocPtc}\), which prepares for the execution of the PTcalc commands 05:-12:. The CafeOBJ’s select command at 04: selects the module \(\text{APPENDassocPtc}\) as the current module. The PTcalc’s :goal command at 05: initiates a proof tree that consists only of the root node, sets CTM(root) = \(\text{APPENDassocPtc}\) and STP(root) = \{eq \text{appendAssoc}(110,120,130) = \text{true} .\}. The :def command at 06:-07: gives the name l1\@ to the case-split (:\text{csp}) command with the exhaustive two equations e1 and e2. Note that the name l1\@ is overloaded to denote the constant l1\@ of sort LIST and the command name, defined with :def, for refining the constant l1\@.

The :apply command at 08: applies the defined case-split command l1\@ at the root node; creates root’s two child nodes 1 and 2 with STP(1) = STP(2) = STP(root), INA(1) = INA(root)∪{e1}, INA(2) = INA(root)∪{e2}; check \(i \in \text{STP}(i)\) for \(i = 1,2\) by applying \text{rd-} command to the nodes (i.e., proof modules) 1,2. The command \text{rd-} is built-in, and computes the reduced form of each element of \text{STP}(\text{gl}) for checking whether \(\text{gl} \in \text{STP}(\text{gl})\). holds. \(i \in \text{STP}(i)\) holds for \(i = 1\) and the node (goal) 1 is discharged, but not for \(i = 2\).

09:-11: gives the name iHyp to the initialize (:\text{init}) command that creates the induction hypothesis equation (3.3.2 04:-05:). An initialization command “:init \text{eq} by \{\text{subst}\}” declares initialization of an equation \text{eq} with a substitution \text{subst}. \text{eq} is declared on-line as (\text{eq} \ldots) (10:) or off-line as [ \text{lbl} ] (4.2.3 21:) where lbl is a label of an already declared equation.

\(^{12}\)01:-12: is in the file list-append.cafe on the Web.
12: applies the command \texttt{iHyp} and \texttt{rd-} to the node 2; the application \texttt{iHyp} adds the created equation to \texttt{INA}(2); the application \texttt{rd-} checks that 2 \models \texttt{STP}(2) holds, and the proof is over.

4.2. Well-Founded Induction (WFI)

WFI is well recognized as the generic induction scheme that subsumes a variety of induction schemes. In CafeOBJ, WFI nicely coordinates to PTcalc and provides a universal and transparent induction scheme \cite{16}. The WFI in CafeOBJ supports naturally (i) induction with respect to multiple parameters, (ii) simultaneous induction, (iii) induction with associative/commutative constructors.

4.2.1. Principle of WFI

The principle of WFI is well established (e.g., 14.1.5 of \cite{32}, A.1.6 & A.1.7 of \cite{33}), and formulated as follows.

\textbf{Fact 13.} (Principle of WFI) Let \( T \) be a set and \( \text{wf} > \) be a well-founded binary relation on \( T \), i.e., \( \text{wf} > \subseteq T \times T \) and there is no infinite sequence of \( t_i \in T \) \((i = 1, 2, \ldots)\) with \((t_i, t_{i+1}) \in \text{wf} > \). Let \( p \) be a predicate on \( T \) (a function from \( T \) to truth values \{true, false\}), then the following holds, where \( t \text{wf} > t' \) means \((t, t') \in \text{wf} > \).

\[
\forall t \in T[(\forall t' \in T((t \text{wf} > t') \Rightarrow p(t'))) \Rightarrow p(t)] \Rightarrow \forall t \in T(p(t))
\]

That is, if \( p(t) \) whenever \( p(t') \) for all \( t' \in T \) such that \( t \text{wf} > t' \), then \( p(t) \) for all \( t \in T \).

It is easy to see \textbf{Fact 13} holds. Assume there exists \( t \in T \) such that \( \neg (p(t)) \), then there should be \( t' \in T \) such that \( t \text{wf} > t' \) and \( \neg (p(t')) \). Repeating this produces an infinite \( \text{wf} > \)-descending sequence. It conflicts with well-foundedness of \( \text{wf} > \).

4.2.2. WFI Scheme in CafeOBJ

Let \( p(y_1, \ldots, y_n) \in T_Y(Y)_\text{Bool} \) be a Boolean term with finite variables \( Y = \{y_1, \ldots, y_n\} \) that describes a property of interest with \( n \) parameters for a module (specification) \( M = (\Sigma, E) \). The WFI can be applied to the tuple \((y_1, \ldots, y_n)\) of parameters of \( p(y_1, \ldots, y_n) \). Let the sort of \( y_i \) be \( s_i \) for \( 1 \leq i \leq n \), and the set of the tuples \( T_p^s \equiv (T_\Sigma^s_1 \times \cdots \times T_\Sigma^s_n) \) be the set \( T \) of \textbf{Fact 13} \( T_p^s \) is called the set of \( p \)'s \textbf{argument tuples}. Let \( \text{wf} > \) be a well-founded binary relation on \( T_p^s \), \( \tilde{y} \) be the tuple of variables \( y_1, \ldots, y_n \), and \( \tilde{y}' \) be the tuple of variables \( y'_1, \ldots, y'_n \). The \textbf{Fact 13} is rewritten as follows. Note that \( \tilde{y} \) and \( \tilde{y}' \) in this context are tuples of mathematical variables and not of CafeOBJ variables. They are used as variables of both kinds interchangeably in this subsection.

\[
\forall \tilde{y} \in T_p^s((\forall \tilde{y}' \in T_p^s((\tilde{y} \text{wf} > \tilde{y}') \Rightarrow p(\tilde{y}'))) \Rightarrow p(\tilde{y})) \Rightarrow \forall \tilde{y} \in T_p^s(p(\tilde{y}))
\]

Let some appropriate binary relation \( \text{wf} > \subseteq T_p^s \times T_p^s \) be defined on \( M \), and
$M_{wf}$ be the module with the defined relation. Then the following is obtained.

$$(M_{wf} \models ([\forall \bar{y}'((\bar{y} \, wf \, \bar{y}') \implies p(\bar{y}'))) \implies p(\bar{y}))) \Rightarrow M \models p(\bar{y})$$

By applying Fact 3 to the premise ($M_{wf} \models \cdots$) with the correspondence $\bar{y} \mapsto Y$, the following is obtained, where $y_{\bar{a}_i}$ is a fresh constant for the variable $y_i$ ($1 \leq i \leq n$), $Y_{\bar{a}} = \{y_{\bar{a}_1}, \ldots, y_{\bar{a}_n}\}$, $\bar{y}_{\bar{a}} = y_{\bar{a}_1}, \ldots, y_{\bar{a}_n}$.

$$(M_{wf} \cup Y_{\bar{a}} \models ([\forall \bar{y}'((\bar{y} \, wf \, \bar{y}') \implies p(\bar{y}'))) \implies p(\bar{y}_{\bar{a}}))) \Rightarrow M \models p(\bar{y})$$

By applying Fact 4 (iii) with the correspondences: $p \mapsto [\forall \bar{y}'((\bar{y} \, wf \, \bar{y}') \implies p(\bar{y}'))]$, $q \mapsto p(y_{\bar{a}})$ and seeing, via Fact 2 (ii),

$$\text{eq} [\forall \bar{y}'((\bar{y} \, wf \, \bar{y}') \implies p(\bar{y}'))] = \text{true}.$$ is equal to

$$\text{cq} \, p(\bar{y}') = \text{true if } y_{\bar{a}} \, wf \, \bar{y}' .$$ the following is obtained.

**Fact 14. (WFI Scheme in CafeOBJ)**

$$(M_{wf} \cup Y_{\bar{a}} \cup \{\text{cq} \, p(\bar{y}') = \text{true if } y_{\bar{a}} \, wf \, \bar{y}' \}) \models p(y_{\bar{a}})) \Rightarrow M \models p(\bar{y}) \quad \square$$

### 4.2.3. Proof Scores with WFI Scheme

The following CafeOBJ code 01:-23: is an effective proof score for the premise of Fact 14 with the correspondences:

APPENDassoc@$(X,L,\text{LIST}(X)) = M \cup Y_{\bar{a}}$,

APPENDassocWfiHyp$(X,\text{LIST}(X)) = M_{wf} \cup Y_{\bar{a}} \cup \{\text{cq} \, p(\bar{y}') = \text{true if } y_{\bar{a}} \, wf \, \bar{y}' \}$.

---

01: mod LLLwfRl(X :: TRIV,Y :: LIST(X)) { 02: [L11] 03: op t : List List List -> Lll {constr} . 04: pred _wf>_ : Lll Lll . 05: vars L1 L2 L3 : List . var E : Elt . 06: eq t(E | L1,L2,L3) wf> t(L1,L2,L3) = true .} 07: mod APPENDassocWfiHyp 08: (X :: TRIV,Y :: LIST(X)) { 09: ex(APPENDassoc@$(X,Y) + LLLwfRl(X,Y)) 10: vars L1 L2 L3 : List . 11: cq[appWfiHyp :nonexec] : 12: appendAssoc(L1,L2,L3) = true 13: if t(110,120,130) wf> t(L1,L2,L3) . } 14: mod APPENDassocWfiPtc(X :: TRIV) { 15: pr(APPENDassocWfiHyp(X,LIST=\{X\})) 16: op e$ : -> Elt . op li$ : -> List .} 17: select APPENDassocWfiPtc . 18: :goal{eq appendAssoc(110,120,130)= true .} 19: :def 110 = :csp{eq 110 = nil . 20: eq 110 = e$ | li$ .}
Module LLLwfRl (01:-06:) defined the well-founded relation \_wf>\_ (04:) on LLI (3-tuples of List) with the equation 06:. Like the equation 06:, the majority of well-founded relations \_wf>\_ are defined based on the strict sub-term relations on constructor terms. The equation 11:-13: corresponds to “cq p(y') = true if y'\_wf> y'.” of Fact 14 16: declares two constants e$, 11$ for refining the constant 11\@ with PTCalc.

5. Proof Scores for Transition Systems

It is widely recognized that the majority of services and systems in many fields can be modeled as transition systems. A transition system is defined as a 3-tuple \((St, Tr, In)\). \(St\) is a set of states, \(Tr \subseteq St \times St\) is a set of transitions on the states (i.e., a state transition relation), and \(In \subseteq St\) is a set of initial states. \((s, s') \in Tr\) denotes the transition from a state \(s\) to a state \(s'\). A finite or infinite sequence of states \(s_1 s_2 \cdots s_n (1 \leq n)\) or \(s_1 s_2 \cdots\) with \((s_i, s_{i+1}) \in Tr\) for each \(i \in \{1, \cdots, n-1\}\) or \(i \in \{1, 2, \cdots\}\) is defined to be a transition sequence. Note that any \(s \in St\) is defined to be a transition sequence of length 1.

5.1. Invariant Properties

Given a transition system \(TS = (St, Tr, In)\), a state \(s' \in St\) is defined to be reachable iff there exists a transition sequence \(s_1 s_2 \cdots s_n (1 \leq n)\) with \(s_n = s'\) such that \(s_1 \in In\). A state predicate (a function from \(St\) to \(Bool\)) \(p\) is defined to be an invariant (or an invariant property) iff \(p(s') = true\) for any reachable state \(s'\). A state predicate may have an extra data argument with a declaration like “pred St Data”.

Fact 15. (Invariants) \[14] Let init be a state predicate that specifies the initial states, that is, \(\forall s \in St (init(s) \Leftrightarrow s \in In)\). Let \(p_1, p_2, \cdots, p_n (1 \leq n)\) be state predicates, and \(iinv(s) = (p_1(s) \land p_2(s) \land \cdots \land p_n(s))\) for \(s \in St\). The following two conditions are sufficient for \(p_i (1 \leq i \leq n)\) to be an invariant.

\[
\begin{align*}
& (INV1) \ \forall s \in St (init(s) \Rightarrow iinv(s)) \\
& (INV2) \ \forall (s, s') \in Tr (iinv(s) \Rightarrow iinv(s')) \\
\end{align*}
\]

□

A predicate that satisfies the conditions (INV1) and (INV2) like \(iinv\) is called an inductive invariant. If a state predicate \(p\) itself is an inductive invariant then taking \(p_1 = p\) and \(n = 1\) is enough. However, \(p_1, p_2, \cdots, p_n (n > 1)\) are almost always needed to be found for getting an inductive invariant, and to find them is a most difficult part of the invariant verification.

\[14\]Symbols like \(s, s', s_i\) were used to denote sorts \(\mathcal{E}, \mathcal{T}, \mathcal{I}\). The same symbols denote states also, but can be distinguished by the context.
5.2. Observational Transition Systems (OTS)

An equational specification $SP = (\Sigma, E)$, $\Sigma = (S, \leq, F, F^c)$ is called an OTS specification if $SP$ is described as follows and defines a transition system $TS = (St, Tr, In)$. An OTS is defined as an $(S, \leq, F, F^c)$-algebra here and differs from the original treatment in [44].

Assume (i) a unique state sort $St \in S$ and $S' \equiv S \setminus \{St\}$, (ii) a unique initial state constant $init \in (T_2)_{St}$, (iii) a set of operators called actions $Ac \subseteq F^c$ of which each element $a_i \in Ac(1 \leq i \leq m_a)$ has a rank $Sta_1^a \cdots a_m^a St(0 \leq n_a, s_j^a \in S^d, 0 \leq j \leq n_a)$, (iv) a set of operators called observers $Ob \subseteq F$ of which each element $o_i \in Ob(1 \leq i \leq m_o)$ has a rank $Sta_1^o \cdots a_m^o St(0 \leq n_o, s_j^o \in S^d, 0 \leq j \leq n_o, s^o \in S^d)$.

The observers give observable values of a system, and the behavior of the system is described by defining the changes of the observable values before and after the actions with the conditional equations of the following form for $1 \leq i \leq m_a, 1 \leq j \leq m_o, 1 \leq k \leq l_{o/a}$:

$$cq \ o_i(a_j(X^{st}, X^{ao}), X^{ao}) = v_k^{o/a_j}(X^{st}, X^{ao}, X^{ao}) \text{ if } c_k^{o/a_j}(X^{st}, X^{ao}, X^{ao})$$

where

$$X^{ao} = X_1^{ao}, \cdots, X_i^{ao}, X_j^{ao}, X_{i+j}^{ao}, X_j^{ao}, \cdots, X_{m_a}^{ao},$$

each $X$ is a variable of the following sort

$$X^{st}, X^{ao}, s_{n_o}^{ao}(1 \leq h \leq n_o), X_{h}^{ao}, s_{h}^{ao}(1 \leq h \leq n_a),$$

and

$$c_k^{o/a_j}(X^{st}, X^{ao}, X^{ao}) \in T_2(X^{st}, X^{ao}, X^{ao}),$$

Then the transition system $TS = (St, Tr, In)$ is defined as follows.

(i) $St \equiv (T_2)_{St}$

(ii) $Tr \equiv \{(t_0, a_i(t_0, t_1, \cdots, t_{m_a})) | a_i(t_0, t_1, \cdots, t_{m_a}) \in (T_2)_{St}\}$

(iii) $In \equiv \{init\}$

The proof scores for the verification conditions of invariant properties (Fact 15 (INV1) (INV2)) on OTS specifications are well studied from the early age of CafeOBJ [44]. Quite a few significant cases are developed as basic proof scores [12, 13, 20], and fairly non-trivial automatic proofs with CITP (27, 49) are achieved [23, 50].

5.3. (p leads-to q) Properties

Invariants are fundamentally important properties of transition systems. They are asserting that something bad will not happen (safety). However, it is sometimes also important to assert that something good will surely happen (liveness). [43, 46] studied proof scores for liveness properties. This subsection focuses on (p leads-to q) properties and provides a concrete and unified approach for constructing proof scores for liveness properties.
For two state predicates $p$ and $q$, the \textit{(p leads-to q) property} is a liveness property asserting that a transition system will get into a state $s$ with $q(s) = \text{true}$ whenever the system gets into a state $s$ with $p(s) = \text{true}$ no matter what transition sequence is taken. More formally, a transition system is defined to have the \textit{(p leads-to q) property} as follows. A finite transition sequence $s_1 \cdots s_n$ ($1 \leq n$) is called \textbf{terminating} if there is no $s'$ such that $(s_n, s') \in T_r$.

A transition system is defined to have the \textit{(p leads-to q) property} iff for any terminating or infinite transition sequence starting from the state $s$ such that $s$ is reachable and $p(s) = \text{true}$, there exists a state $\hat{s}$ such that $q(\hat{s}) = \text{true}$ somewhere in the sequence.

The \textit{(p leads-to q) property} is adopted from the UNITY logic \cite{7}, the above definition is, however, not the same as the original one. In the UNITY logic, the basic model is the parallel program with parallel assignments, and \textit{(p leads-to q) property} is defined through applications of inference rules.

\textbf{Fact 16.} (\textit{p leads-to q}) \cite{14,15} Let $TS = (St, Tr, In)$ be a transition system, $p$, $q$, $inv$ be state predicates with $inv$ being a $TS$’s invariant property, and $m$ be a function from $St$ to the set of natural numbers. If the followings (LT1) and (LT2) are proved by choosing $inv$ and $m$ appropriately, then $TS$ has the \textit{(p leads-to q) property}.

\begin{enumerate}
\item[(LT1)] $\forall (s, s') \in Tr$ 
\hspace{1cm} $((inv(s) \land p(s) \land \neg q(s)) \Rightarrow ((p(s') \lor q(s')) \land (m(s) > m(s'))))$
\item[(LT2)] $\forall s \in St$ 
\hspace{1cm} $((inv(s) \land p(s) \land \neg q(s)) \Rightarrow (\exists s' \in St((s, s') \in Tr)))$
\end{enumerate}

\textbf{□}

A \textit{(p leads-to q) property} sometimes needs to be proved in multiple steps by making use of the following facts which are easily obtained from the definition of \textit{(p leads-to q)}. \par

\textbf{Fact 17.} (Multiple leads-to) Let $(p \bowtie q)$ stand for \textit{(p leads-to q)} and $p, q, r$ be state predicates, then the followings hold.

\begin{enumerate}
\item[(MLT1)] $(p \bowtie q) \land (q \bowtie r) \Rightarrow (p \bowtie r)$
\item[(MLT2)] $(p \bowtie q) \lor (p \bowtie r) \Rightarrow (p \bowtie q \lor r)$
\item[(MLT3)] $(p \bowtie r) \land (q \bowtie r) \Rightarrow (p \lor q \bowtie r)$
\end{enumerate}

\textbf{□}

5.4. Transition Specifications and Builtin Search Predicates \cite{14}

A natural way to define a transition system $TS = (St, Tr, In)$ is to define transitions with rewrite rules \cite{2.6.2}. The CafeOBJ’s builtin search predicates facilitate verifications of transition systems with the rewriting transitions.\footnote{The ideas underlying this subsection were first presented in \cite{14}. The content of this subsection is much more elaborated based on the unified theories described in Section \cite{2}.}
5.4.1. Transition Specifications

Let $\text{State}$ be a special sort that satisfies (i) there is no supersedor $\mathbb{2}[1.1]$ of $\text{State}$ (ii) no term of the sort $\text{State}$ has a strict subterm of the sort $\text{State}$. A sort like $\text{State}$ is called topmost sort $\mathbb{2}[2.4]$, and the quotient set $(T_{\Sigma,E\cup AC})_{\text{State}}$ $\mathbb{2}[2.5]$ is supposed to model the state space of the system specified, where $AC$ is the AC-equations for the AC operators in $\Sigma$ $\mathbb{2}[2.6]$.

Let “cq $l(X) \Rightarrow r(X)$ if $c(X)$.” be a rewrite rule on the sort $\text{State}$, i.e., $l(X), r(X) \in (T_{\Sigma}(X))_{\text{State}}$, then “ctr $l(X) \Rightarrow r(X)$ if $c(X)$.” is a transition rule. Let $Tl$ be a set of transition rules then a 3-tuple $(\Sigma, E, Tl)$ defines one-step transition relation and transition relation $\rightarrow_{AC} \subseteq (T_{\Sigma})_{\text{State}} \times (T_{\Sigma})_{\text{State}}$ by the following inference rules, where $t_i, t_r, t_m, \in (T_{\Sigma})_{\text{State}}$ and $\rightarrow_{st}$ is defined by $(\Sigma, E)$ $\mathbb{2}[2.6.2]$, 

(a) If $(t_l =_{AC} t_r)$, $\Rightarrow t_l \rightarrow_{st} t_r$.
(b) If $(\neg(t_l =_{AC} t_r))$, $t_l \rightarrow_{st} t_m \land t_m \rightarrow_{st} t_r \Rightarrow t_l \rightarrow_{st} t_r$.
(c1) For a transition rule “ctr $l(X) \Rightarrow r(X)$ if $c(X)$.” in $Tl$, a valuation $\theta : X \rightarrow T_{\Sigma}$, if $t_l =_{AC} \theta(l(X)) \land t_r =_{AC} \theta(r(X))$, $\theta(c(X)) \rightarrow_{\text{true}} t_l \rightarrow_{st} t_r$.

CafeOBJ implements one-step weak transition relation and weak transition relation $\rightarrow_{st} \subseteq \rightarrow_{st}$ instead of $\rightarrow_{AC} \subseteq \rightarrow_{st}$. Inference rules (a), (b) for $\rightarrow_{st} \subseteq \rightarrow_{st}$ are the same as those for $\rightarrow_{AC} \subseteq \rightarrow_{st}$. The rule (c1) is as follows, where $t_l \in (T_{\Sigma})_{\text{State}}$, $\text{red}$ and $\rightarrow_{\text{true}}$ are defined by $(\Sigma, E)$ $\mathbb{2}[2.6.3]$. Note that the left hand side $l(X)$ is AC-matched with the reduced term $t_l^{\text{red}}$.

(c1) For a transition rule “ctr $l(X) \Rightarrow r(X)$ if $c(X)$.” in $Tl$, a valuation $\theta : X \rightarrow T_{\Sigma}$, if $t_l^{\text{red}} =_{AC} \theta(l(X))$, $\theta(c(X)) \rightarrow_{\text{true}} t_l \rightarrow_{st} \theta(r(X))$.

A transition rule in $Tl$ of a 3-tuple $(\Sigma, E, Tl)$ is called left reduced if its left hand side $l(X)$ is reduced modulo AC in $(\Sigma, E)$ $\mathbb{2}[2.6.3]$ by considering $X$ as a set of fresh constants. A set of transition rules $Tl$ is called left reduced if each element is so. Each effective transition rule is assumed to be left reduced in CafeOBJ. This assumption is natural, for “st(5) => final” is better and safer than “st(2 + 3) => final”.

The relation $\rightarrow_{st}$ naturally defines the state transition relation: $\rightarrow_{st} \subseteq (T_{\Sigma,E\cup AC})_{\text{State}} \times (T_{\Sigma,E\cup AC})_{\text{State}}$ by defining

$\exists [st_1]_{E\cup AC} \Rightarrow r_{(T_{\Sigma})_{\text{State}}} [st_2]_{E\cup AC}$ iff $\exists t'_1, t'_2 (st_1 =_{E\cup AC} st'_1 \land st_2 =_{E\cup AC} st'_2 \land st'_1 \rightarrow_{st} st'_2)$.

Let $In \subseteq (T_{\Sigma})_{\text{State}}$; then a 4-tuple $TS = (\Sigma, E, Tl, In)$ is called a transition specification. A transition specification defines two transition systems: (ST1) $((T_{\Sigma,E\cup AC})_{\text{State}}, \rightarrow_{AC}, \{[st]_{E\cup AC} \mid st \in In\})$, (ST2) $((T_{\Sigma})_{\text{State}}, \rightarrow_{st}, In)$.

$\uparrow$In $\mathbb{2}[2.7]$ a topmost sort $ts$ is defined to be (i) a topmost sort of a connected component of order-sorted sorts, such that (ii) no operator has $ts$ as the sort of any of its arguments.
(ST1) is an ideal system and cannot be implemented, and CafeOBJ implements (ST2). \([st] \equiv_{E \cup AC}\) consists of all the terms \(st'\) such that \(st' \equiv_{E \cup AC}\) \(st\), and if \((\Sigma, E)\) is red-complete (Fact 10), each \([st] \equiv_{E \cup AC}\) can be represented by the term \(st'_{\text{red}}\) for any \(st' \in [st] \equiv_{E \cup AC}\) modulo AC.

It is easy to see from the above definition (c1) and \(\Sigma \equiv_{E, AC}\) that \(\rightarrow_{TL, AC}\) is complete enough such that

\[(Tl, AC) \forall t, u \in T_{\Sigma}(t \rightarrow_{TL, AC} u \Rightarrow \exists u' \in T_{\Sigma}(t \rightarrow_{TL, AC} u' \land u'_{\text{red}} =_{AC} u_{\text{red}})).\]

Hence, the following bi-simulation relation is obtained.

\[(BS) ([st] \equiv_{E \cup AC} \rightarrow_{TL, AC} [st]_{\text{red}}) \iff \exists st'_{2} \in [st] \equiv_{E \cup AC} \exists st'_{1} \rightarrow_{TL, AC} st'_{1} \land st'_{1} \equiv_{E \cup AC} st'_{2} \land st'_{2} \equiv_{E \cup AC} st'_{1} \text{red}.)\]

Invariant/leads-to properties of a transition system (ST1) are verified by constructing proof scores for the corresponding (ST2).

A verification condition Fact 15 (INV1) can be proved without using the builtin search predicates. Verification conditions Fact 15 (INV2), Fact 16 (LT1) (LT2), however, need builtin search predicates that search all possible transitions possible in (ST2), and that implies to search all possible transitions possible in (ST1) thanks to (BS).

5.4.2. Builtin Search Predicates

The primary builtin search predicate is declared as follows.

\[
\text{pred \_=(\star,1) => \_ if \_suchThat \_} : \text{State State Bool Bool Info}.
\]

Info is a sort for outputting information; the 1st argument is the current state \(s \in (T_{\Sigma})_{\text{State}}\); the 2nd and 3rd arguments are variables \(SS: \text{State}\) and \(CC: \text{Bool}\) for binding the next state and the condition to be found, respectively; the 4th argument is a predicate \(p(s, SS, CC)\) whose validity is to be checked; the 5th argument is a term \(i(s, SS, CC)\) for outputting the information.

For a state term \(s \in (T_{\Sigma})_{\text{State}}\) the CafeOBJ’s reduction of a Boolean term:

\[
s \equiv_{\star,1} \Rightarrow SS: \text{State if } CC: \text{Bool}
\]

suchThat \(p(s, SS, CC)\) \{i(s, SS, CC)\}

in a 3-tuple \(M = (\Sigma, E, El)\) behaves as follows.

(i) Search for every pair \((tl, \theta)\) of a transition rule \(tl = \text{ctr}(X) \Rightarrow r(X)\) if \(c(X)\) in \(Tl\) and a valuation \(\theta : X \rightarrow T_{\Sigma}\) such that \(s =_{AC} \theta(l(X))\).

(ii) For each found pair \((tl, \theta)\), let \((SS = \theta(r(X)))\) and \((CC = \theta(c(X)))\) and print out \(i(\theta(l(X)), \theta(r(X)), \theta(c(X)))\) and \(tl\) if \(p(\theta(l(X)), \theta(r(X)), \theta(c(X))\)) \text{red} = \text{true}\).

(iii) Returns \text{true} if some print out exists, and returns \text{false} otherwise.

\(p(s, SS, CC)\) can be defined to check whether a relation between current state \(s\) and next state \(SS\) holds. Let “\text{pred cnr : State State Data}.” (Current Next Relation) be a predicate stating some relation like one in Fact 15 (INV2) or Fact 16 (LT1) of the current and next states. \text{Data} may be necessary if it appears in some state predicates involved in the current/next state relation.

Let check-cnr be defined as follows.

\[
\text{pred check-cnr : State Data}.
\]

\[
\text{eq check-cnr}(S: \text{State}, D: \text{Data}) = \not (S = \star,1) \Rightarrow SS: \text{State if } CC: \text{Bool}
\]

28
suchThat not((CC implies cnr(S,SS,D)) == true) 
{i(S,SS,CC,D)}.

Since reducing $(S =(*,1)=>+ SS:State$ if $CC:Bool ...)$ searches all the possible transitions from $S$, “(check-cnr(S:State,D:Data))$red $= true$” means “for any transition from $S ((CC$ implies $cnr(S,SS,D)) == true) = true)”, and the following fact is obtained.

**Fact 18.** For each $s\in(TS)State$ and each $d\in(TD)Data$ of a 3-tuple $M = (\Sigma, E, El)$, $M \vdash check-cnr(s, d) \Rightarrow \forall (s, s')\in\rightarrow_{\text{TLAC}}(cnr(s, s', d) = true)$.

There is another builtin search predicate:

pred _=(1,1)=>+_ : State State .

Reducing $(s = (1,1)=>+ SS:State)$ searches all the possible one-step transitions from a state $s$ and returns $true$ if found. Hence the following fact.

**Fact 19.** For each $s \in (T\Sigma)State$ of a 3-tuple $M = (\Sigma, E, El)$, $M \vdash (s = (1,1)=>+ SS:State) \Rightarrow \exists (s, s')\in\rightarrow_{\text{TLAC}}$.

Proof scores for verification conditions (INV2), (LT1) and (LT2) can be constructed by making use of Fact 18 and Fact 19 respectively.

PTcalc+WFI proof scores for leads-to properties of a practical size cloud protocol in transition rules have been developed by making use of Fact 15, Fact 16, Fact 18 and Fact 19 in a follow up research of [57].

### 6. QLOCK Example

This section is devoted to show how to construct proof scores for transition systems based theories and methods presented in Sections 2, 3, 4, 5 by making use of a simple but non-trivial example QLOCK (Queue LOCKing) of mutual exclusion protocols. Mutual exclusion protocol is described as follows:

Assume that many agents (or processes) are competing for a common equipment (e.g., a printer or a file system), but at any moment of time only one agent can use the equipment. That is, the agents are mutually excluded in using the equipment. A protocol (distributed mechanism or algorithm) which can achieve the mutual exclusion is called a **mutual exclusion protocol**.

QLOCK is realized by using a global queue (first in first out storage) of agent names (or identifiers) as follows:

- Each of unbounded number of agents who participates in the protocol behaves as follows:

  [want] If the agent wants to use the common equipment and its name is not in the queue yet, put its name at the bottom of the queue.
If the agent wants to use the common equipment and its name is already in the queue, check if its name is on the top of the queue. If its name is on the top of the queue, start to use the common equipment. If its name is not on the top of the queue, wait until its name is on the top of the queue.

If the agent finishes to use the common equipment, remove its name from the top of the queue.

- The protocol starts from the state with the empty queue.

6.1. QLOCK as an Observational Transition System (OTS)

QLOCK can be described as an OTS specification \(\text{QLOCK/OTS}\) and the specification is called QLOCK/OTS. For an OTS specification, (i) no need to design structure for state space, (ii) no need to make some part red-complete for proof scores to be valid, but to prove leads-to properties is difficult.

6.1.1. QLOCK/OTS: System Specification of QLOCK as an OTS

For QLOCK/OTS and proof scores for that, the specification QUEhdTlPt (24:-29:) of generic (or parameterized) queues with hd (head), tl (tail), pt (put) operators is necessary. QUEhdTlPt adds the definitions of operators tl_ and pt_ to QUEhd (18:-22:) that adds the definition of operator hd_ to QUE (14:-16:).

QUE is defined by renaming a sort and operators of SEQ= (18:-16:) with *{...}.

SEQ= enhances the builtin operator _.=_ of SEQ that defines generic sequences. Note that hd_ takes the left-most element of a queue, and pt_ puts a element to a queue from the right.

```plaintext
01: -- generic SEQuences
02: mod! SEQ (X :: TRIV) { [Elt < Seq] -- an element is a sequence
03: op nil : -> Seq {constr} .
04: op __ : Seq Seq -> Seq {constr assoc id: nil} . }
05: -- enhancing the builtin operator _.=_ of SEQ
06: mod! SEQ= (X :: TRIV) { pr(SEQ(X))
07: eq (nil = (E:Elt S2:Seq)) = false .
08: cq ((E1:Elt S1:Seq) = (E2:Elt S2:Seq)) =
09: ((E1 = E2) and (S1 = S2))
10: if not((S1 == nil) and (S2 == nil)) . }
11: -- generic queues
12: mod! QUE (X :: TRIV) {
13: op nil -> nilQ, op __ -> _|_} }
14: mod! QUEhd (X :: TRIV) {
15: pr(SEQ=)(sort Seq -> Que,
16: op nil -> nilQ, op __ -> _|_)) }
17: -- generic queues with hd (head) operator
18: mod! QUEhd (X :: TRIV) {
```

\[\text{ QUEhdTlPt (24:-29:) is in the files seq-spc.cafe, que-spc.cafe, ots-sys-spc.cafe on the Web: https://cafeobj.org/~futatsugi/misc/apsco-220907/}\]
19:  pr(QUE(X)) [Elt < EltEr]
20:  op hd_: Que -> EltEr.
21:  eq hd(X:Elt | Q:Que) = X.
22:  eq (hd(nilQ) = X:Elt) = false. }
23:  --> generic queues with hd (head), tl (tail), pt (put) operators
24:  mod! QUEhdTlPt (X :: TRIV) {
25:    pr(QUEhd(X))
26:  op tl_ : Que -> Que.
27:  eq tl(X:Elt | Q:Que) = Q.
28:  op pt : Elt Que -> Que.
29:  eq pt(X:Elt,Q:Que) = Q | X. }
30:  --> generic queues with hd (head), tl (tail), pt (put) operators
31:  mod! LABEL {
32:    [LabelLtl < Label]
33:    ops rm wt cs : -> LabelLtl {constr} .
34:    -- var L : Label .
35:    -- eq (L = L) = true . -- instance of builtin equation
36:    vars L1 L2 : LabelLtl .
37:    eq (L1 = L12) = (L11 == L12) . }
38:  --> generic queues with hd (head), tl (tail), pt (put) operators
39:  mod* AID {
40:    --> Agent Identifiers
41:    mod* AID {[Aid]}  
42:    --> generic queues with hd (head), tl (tail), pt (put) operators
43:    mod! QLOCK/OTS (X :: AID) {
44:      --> generic queues with hd (head), tl (tail), pt (put) operators
45:      pr(LABEL)
46:      pr(QUEhdTlPt(X)*{sort EltEr -> AidEr})
47:      --> generic queues with hd (head), tl (tail), pt (put) operators
48:      -- state space of system
49:      [St]  
50:    --> generic queues with hd (head), tl (tail), pt (put) operators
51:  }
47: op init : -> St {constr} .
48: op want : St Aid -> St {constr} .
49: op try : St Aid -> St {constr} .
50: op exit : St Aid -> St {constr} .

QLOCK is specified by defining the values of the observers pc and que
(52:-53:) for each term of the sort St. 55:-56: defines the values of pc and
que for the term init representing initial states. 60:-64: defines the values of
pc and que for each term with the top operator want. 66:-70: is for try and
72:-76: is for exit.

51: -- observers of system
52: op pc : St Aid -> Label .
53: op que : St -> Que .

54: -- for each initial state
55: eq pc(init,I:Aid) = rm .
56: eq que(init) = nilQ .

57: -- variables
58: var S : St . vars I J : Aid .

59: -- for want
60: ceq pc(want(S,I),J) = (if I = J then wt else pc(S,J) fi)
61: if (pc(S,I) = rm) .
62: ceq pc(want(S,I),J) = pc(S,J) if not(pc(S,I) = rm) .
63: ceq que(want(S,I)) = pt(I,que(S)) if (pc(S,I) = rm) .
64: ceq que(want(S,I)) = que(S) if not(pc(S,I) = rm) .

65: -- for try
66: ceq pc(try(S,I),J) = (if I = J then cs else pc(S,J) fi)
67: if (pc(S,I) = wt and hd(que(S)) = I) .
68: ceq pc(try(S,I),J) = pc(S,J)
69: if not(pc(S,I) = wt and hd(que(S)) = I) .
70: eq que(try(S,I)) = que(S) .

71: -- for exit
72: ceq pc(exit(S,I),J) = (if I = J then rm else pc(S,J) fi)
73: if (pc(S,I) = cs) .
74: ceq pc(exit(S,I),J) = pc(S,J) if not(pc(S,I) = cs) .
75: ceq que(exit(S,I)) = tl(que(S)) if (pc(S,I) = cs) .
76: ceq que(exit(S,I)) = que(S) if not(pc(S,I) = cs) .}

6.1.2. Properties of Interest on QLOCK/OTS and WFI Hypotheses for Them

The main property of interest on QLOCK/OTS is the mutual exclusion property
mtx(S:St,I:Aid,J:Aid) defined in 07:-09:. However,

OTSmxInvPrp = mtx(S:St,I:Aid,J:Aid)

is difficult to prove by induction, and qtp(S,K:Aid) (11:-12:) is needed to be
found to make

OTSmxInvPrp = (mtx(S:St,I:Aid,J:Aid) and qtp(S,K:Aid))

provable by induction. That is, (mtx(S:St,I:Aid,J:Aid) and qtp(S,K:Aid))
is an inductive invariant (see [5,7], but mtx(S:St,I:Aid,J:Aid) is not.

2001:-32: is in the files ots-mx-inv-prp.cafe, ots-mx-wfi-hypo.cafe on the Web.
01: --> module defining predicates mtx and qtp
02: mod OTSmxInvPrp {
03: pr(QLOCK/OTS)
04: -- variables
05: vars S : St . vars I J K : Aid .
06: -- predicate defining mutual exclusion (mtx) property
07: pred mtx : St Aid Aid .
08: eq mtx(S,I,J) =
09: ((pc(S,I) = cs) and (pc(S,J) = cs)) implies (I = J) .
10: -- predicate defining queue top (qtp) property
11: pred qtp : St Aid .
12: eq qtp(S,K) = (pc(S,K) = cs) implies (hd(que(S)) = K) .}

The module OTSxWfiHypo (13:-32:) corresponds to the module in the premise of Fact 14 with the following correspondences:
(1) OTSxInvPrp ⊩ M,
(2) S:St,I:Aid,J:Aid,K:Aid = y, y′,
(3) s@,i@,j@,k@$ = y@$,
(4) mtx(S:St,I:Aid,J:Aid) and qtp(S,K:Aid) ⊩ p(y),
(5) 16:-26: = M_wf∪Y@,
(6) 28: + 30: = cq p(y′) = true if y@$ wf> y′ ,
(7) OTSxWfiHypo ⊩ M_wf∪Y@∪{cq p(y′) = true if y@$ wf> y′ }.

(4) shows that p(y) is a conjunction of two conjuncts mtx(S:St,I:Aid,J:Aid) and qtp(S,K:Aid), and (6) shows that the WFI hypothesis for p(y) is divided into two WFI hypotheses 28: for mtx and 30: for qtp. Note that wf> on the argument tuples of p(y)
(s@,i@,j@,k@$) wf> (S:St,I:Aid,J:Aid,K:Aid)
is defined by “s@ wf> S:St” that is defined in 24:-26;.

OTSxWfiHypo is a typical example of a module defining WFI hypotheses for simultaneous induction.

Fact 14 shows that
OTSxWfiHypo ⊩ (mtx(s@,i@,j@) and qtp(s@,k@$))
is a sufficient condition for
OTSxInvPrp ⊩ (mtx(S:St,I:Aid,J:Aid) and qtp(S,K:Aid)).

13: --> proof module defining WFI hypotheses
14: mod OTSxWfiHypo {
15: -- the module on which this WFI applies
16: pr(OTSxInvPrp)
17: -- fresh constants for declaring goal and WFI hypotheses
18: -- goal proposition: “mtx(s@,i@,j@) and qtp(s@,k@$)”
19: op s@ : -> St . ops i@ j@ k@$ : -> Aid .
20: -- variables
21: var S : St . vars H I J K : Aid .
22: -- well-founded binary relation on St
23: pred _wf>_ : St St .
24: eq want(S,H) wf> S = true .
25: eq try(S,H) wf> S = true .
26: eq exit(S,H) wf> S = true .
27: -- WFI hypotheses for mtx
28: c fq[mtx-hypo :nonexec]: mtx(S,I,J) = true if s@ wf> S .
29: -- WFI hypotheses for qtp
6.1.3. PTcalc Proof Scores on OTS\textsubscript{mx}WfiHypo

01:-29: is a proof score with PTcalc for 
\(\text{OTS}\text{mx}WfiHypo \models \text{mtx}(s\@,i\@,j\@)\)
and a similar proof score\(^{21}\) is made for 
\(\text{OTS}\text{mx}WfiHypo \models \text{qtp}(s\@,k\@)\)
and together prove

\(\text{OTS}\text{mx}WfiHypo \models (\text{mtx}(s\@,i\@,j\@) \land \text{qtp}(s\@,k\@))\)

all at once, but the proof score\(^{22}\) tends to be complex.

04:-13: define case-split commands. \texttt{c-s@iwte} (04:-07:) classifies the 
terms of sort \texttt{St} via the four action operators (constructors), and is exhaustive 
(see \[3\]) \texttt{:csp\{eq l = r . eq (l = r) = false .\}} is an abbreviation of \texttt{:csp\{eq l = r . eq (l = r) = false .\}} and is exhaustive for any \(l, r\). The equations defining the 
case-split in 08:-13: appear in the conditions of the conditional equations or 
if\_then\_else\_fi operators in QLOCK/OTS \[6.1.59:-76:0\].

14:-18: define initialization commands each of them initializing a WFI 
hyothesis \[\text{mtx-hypo} \text{6.1.2-28:}\] or \[\text{qtp-hypo} \text{6.1.2-30:}\].

\[01: \text{-- root} \]
\[02: \text{-- executing proof of "\text{mtx}(s\@,i\@,j\@) = true" with PTcalc} \]
\[03: \text{select \text{OTS}\text{mx}WfiHypo} . \]
\[04: \text{:goal\{eq mtx(s\@,i\@,j\@) = true .\}} \]
\[05: \text{:def c-s@iwte = :csp\{eq s\@ = init .} \]
\[06: \text{eq s\@ = want(s\$,h\$) .} \]
\[07: \text{eq s\@ = try(s\$,h\$) .} \]
\[08: \text{eq s\@ = exit(s\$,h\$) .} \]
\[09: \text{:def c-s$h$rm = :ctf\{eq pc(s$,h$) = rm .\}} \]
\[10: \text{:def c-s$h$wt = :ctf\{eq pc(s$,h$) = wt .\}} \]
\[11: \text{:def c-s$h$cs = :ctf\{eq pc(s$,h$) = cs .\}} \]
\[12: \text{:def c-i@=h$- = :ctf\{eq i$ = h$ .\}} \]
\[13: \text{:def c-j@=h$- = :ctf\{eq j$ = h$ .\}} \]
\[14: \text{:def i-s$I-J- = :init as mxs$I-J- [\text{mtx-hypo}] by \{S:St < s$;\}} \]
\[15: \text{:def i-qts$i@ = :init as qts$i@ [\text{qtp-hypo}] by \{S:St < s$; K:Aid < i$;\}} \]
\[16: \text{:def i-qts$j@ = :init as qts$j@ [\text{qtp-hypo}] by \{S:St < s$; K:Aid < j$;\}} \]
\[23\text{01:-29: is in the file ots-mx-psc.cafe on the Web.}\]
\[22\text{In the file ots-mx-2cj-psc.cafe on the Web.}\]
\[23\text{01:-29: is in the file ots-mx-psc.cafe on the Web.}\]
Example 7. (Guessing Next Command)

Construction of effective (i.e., able to discharge all goals) proof scores is basically by trial and error. However, the next command to be applied has a good chance to be indicated by the information gotten from the next target goal (\[4.1.1\]). After executing \texttt{:apply} command at 25:, for example, the reduced form of the goal proposition $\text{mtx}(s@,i@,j@)$ (03:) is detected to be $(\text{true xor } (\text{pc}(s@,j@) = \text{cs}))$ by using PTcalc’s “:\texttt{show goal}”, \texttt{:red} commands, that is exactly the proposition gotten by applying the \texttt{:init} command as defined by $\texttt{i-qts}$ (17:-18:, 26:). Similar guessing is possible at each “next target goal”. □

After executing 01:-29:, inputting the PTcalc command “:\texttt{show proof}” gives the following output c01:-c34: showing the effective proof tree constructed. root (c01:) is the root node of the proof tree. 1 (c01:), 2 (c03:), 3 (c11:), 4 (c25:) are four child nodes (goals) (see \[4.1.1\]) constructed with the case-split command \texttt{c-s$\#$wte}. Each node of the tree except root is represented by a line containing a command name that creates the node and the node name like “[i-s$I-J-$] 2-1*” (c04:). The symbol * after a node name indicates that the node is discharged.

For example, it can be seen that the node 2-1-2 (c10:) is created as 2nd child of the case-split $\texttt{c-s$\#$r}$ of the child of the initialization $\texttt{i-s$I-J-}$ (2-1, c04:) of the 2nd child of the case-split $\texttt{c-s@iwte}$ (2, c03:), and discharged by \texttt{rd-} after $\texttt{c-s$\#$rm}$ in the arguments of the \texttt{:apply} command in 23:. It can also be seen that c18: is discharged with 26: and c21: is discharged with 27:.

The output c13:-c18: shows a major utility of PTcalc. c18: [i-qts] 3-1-1-1-1-2-1* tells that from the goal 3-1 after 4 case-splits $[\text{c-s$h$wt}]$-1, $[\text{c-tqs$h$}]$-1, $[\text{c-i@=h$-$}]$-1, $[\text{c-j@=h$-$}]$-2, and 1 instantiation $[\text{i-qts}$]@-1, the goal 3-1-1-1-1-2-1 is discharged. The checks with the 4 case-splits are achieved with one \texttt{:apply} command:

\begin{verbatim}
25: :apply(c-s$h$wt rd- c-tqs$h$ rd- c-i@=h$-$ c-j@=h$-$ rd-).
\end{verbatim}

The argument of this \texttt{:apply} command is the sequence of 4 :\texttt{csp} commands and 3 \texttt{rd-} commands. This \texttt{:apply} command has the power of “generating and checking” all of the 16 = $2^4$ cases determined by the 4 :\texttt{csp} commands. The full generation (i.e., generation of $2^4$ goals) is exponentially costly and, of course, should be avoided. That is, try to make each \texttt{rd-} command discharge at
least one goal. 25:-27: constructs the effective proof tree c13:-c24: PTcalc commands are fairly simple and succinct comparing with open...close style, and are great help for constructing effective proof trees by trial and error.

c01: root* c11: [c-s@iwte] 3* c25: [c-s@iwte] 4*
c02: [c-s@iwte] 1* c12: [i-s$I-J-] 3-1* c26: [i-s$I-J-] 4-1*
c03: [c-s@iwte] 2* c13: [c-s$h$rm] 3-1-1* c27: [c-s$h$rm] 4-1-1*
c04: [c-s$h$rm] 2-1-* c14: [c-tqs$h$] 3-1-1-1* c28: [c-tqs$h$] 4-1-1-1*
c05: [c-tqs$h$] 2-1-1* c15: [c-j@=h$-] 3-1-1-1-1* c29: [c-j@=h$-] 4-1-1-1-1*
c06: [c-j@=h$-] 2-1-1-1* c16: [i-qts$j@] 3-1-1-1-1-1* c30: [i-qts$j@] 4-1-1-1-1-1*
c07: [i-s$I-J-] 1* c17: [c-j@=h$-] 3-1-1-1-1-2* c31: [c-j@=h$-] 4-1-1-1-2*
c08: [c-j@=h$-] 2-1-1-2* c18: [c-tqs$h$] 3-1-1-1-2* c32: [c-tqs$h$] 4-1-1-2*
c09: [c-tqs$h$] 2-1-1-2-1* c19: [c-s$h$cs] 3-1-1-1-2* c33: [c-s$h$cs] 4-1-1-2-1*
c10: [c-s$h$cs] 2-1-2* c20: [c-j@=h$-] 3-1-1-2-1* c34: [c-j@=h$-] 4-1-2*
c11: [c-s@iwte] 3* c21: [i-qts$j@] 3-1-1-2-1-1* c35: [i-qts$j@] 4-1-2-1*
c12: [i-s$I-J-] 4-1* c22: [c-j@=h$-] 3-1-1-2-2* c36: [c-j@=h$-] 4-1-2-2*
c13: [c-s$h$wt] 3-1-1-1* c23: [c-s$h$wt] 3-1-1-2* c37: [c-s$h$wt] 4-1-2*
c14: [c-tqs$h$] 3-1-1-1-1* c24: [c-tqs$h$] 3-1-1-2* c38: [c-tqs$h$] 4-1-2*
c15: [c-j@=h$-] 3-1-1-1-1-1* c25: [c-s@iwte] 4* c39: [c-s@iwte] 5-1-
c16: [c-j@=h$-] 3-1-1-1-1-2* c26: [c-j@=h$-] 4* c40: [c-j@=h$-] 5-1-1*
c17: [i-qts$j@] 3-1-1-1-1-2-1* c27: [i-qts$j@] 4-1* c41: [i-qts$j@] 5-1-1-1*
c18: [c-tqs$h$] 3-1-1-1-2* c28: [c-tqs$h$] 4-1-1* c42: [c-tqs$h$] 5-1-1-1*
c19: [c-s$h$cs] 3-1-1-2* c29: [c-s$h$cs] 4-1-1-1* c43: [c-s$h$cs] 5-1-1-1-1*
c20: [c-j@=h$-] 3-1-1-2-1* c30: [c-j@=h$-] 4-1-1-2* c44: [c-j@=h$-] 5-1-1-1-1*
c21: [i-qts$j@] 3-1-1-2-1-1* c31: [i-qts$j@] 4-1-1-2-1* c45: [i-qts$j@] 5-1-1-1-1-1*
c22: [c-j@=h$-] 3-1-1-2-2* c32: [c-j@=h$-] 4-1-1-2-2* c46: [c-j@=h$-] 5-1-1-1-1-2*
c23: [c-s$h$wt] 3-1-2* c33: [c-s$h$wt] 4-1-2-1* c47: [c-s$h$wt] 5-1-1-1-2*
c24: [c-s$h$wt] 3-1-2* c34: [c-s$h$wt] 4-1-2* c48: [c-s$h$wt] 5-1-1-2*
c25: [c-s@iwte] 4* c35: [c-s@iwte] 5-1* c49: [c-s@iwte] 5-1-1*

(1) The reduced Boolean value of the goal proposition, e.g., mtx(s@,i@,j@) in 03:; at each node of a proof tree gives many good hints on which command should be applied next (see Example 7). (2) Besides, the possible commands, e.g., commands defined in 04:-18: can be constructed systematically once the set of fresh constants, e.g., 3.1.2-19:,32: is fixed. 50: reported a system that fully automated proofs of invariant properties on OTS based on (1) and (2). The full automation techniques could be applied for automatically generating effective advanced proof scores like 01:-29:.

6.2. QLOCK in Transition Specification (TSP)

QLOCK is described as a transition specification (5.4.1) and the specification is called QLOCK/TSP. As a matter of fact, QLOCK/TSP does not include the specification of initial states In of (Σ, E, Tl, In) and is just a 3-tuple (Σ, E, Tl) (5.4.1). The initial states are specified later. For a 3-tuple (Σ, E, Tl), (i) data structure for the sort State needs to be designed, (ii) (Σ, E) needs to be red-complete (Fact 10) for proof scores with builtin search predicates to be valid, but proof scores for complex properties like leads-to properties can be constructed thanks to transparent specifications of transitions in transition rules.

6.2.1. QLOCK/TSP: System Specification of QLOCK in TSP

For describing QLOCK/TSP, the sort State should be defined so that state transitions can be defined with transition rules on State. For defining State, generic sets with the operators _in_, _=_-, _^_- are specified as the module SET=^ (24:-31:) after the modules SET (01:-06:), SETin (07:-14:), SET= (15:-23:). The assoc and comm attributes of the operator _^_- can be proved (e.g., 15:) and declaration of {assoc comm} (27:) is justified. SETlem (32:-40:) declares necessary lemmas on SET=^.

24: Proof scores for the lemmas are in the file set-lem.cafe on the Web: https://cafeobj.org/~futatsugi/misc/apsco-220907/.
25: 01:-80: is in the files set-psc.cafe, tsp-state-psc.cafe, tsp-sys-psc.cafe on the Web.
01: --> generic sets
02: mod! SET(X :: TRIV) {
03: [Elt < Set]  
04: op empty : --> Set {constr} .
05: op __ : Set Set --> Set {constr assoc comm id: empty} .
06: ceq (S:Set S) = S if not(S == empty) . }

07: --> generic sets with _in_ predicate
08: mod! SETin (X :: TRIV) {
09: pr(SET(X))
10: pred _in_ : Elt Set .
11: eq (E:Elt in empty) = false .
12: eq E1:Elt in (E2:Elt S:Set) = (E1 = E2) or (E1 in S) .
13: eq[in-fls :nonexec]:
14: ((E:Elt in S:Set) and (ES:Set = E S)) = false .

15: --> generic sets with _=_ predicate
16: mod! SET= (X :: TRIV) {
17: pr(SETin(X))
18: pred _=_ : Set Set . -- equal of less
19: eq S:Set =< S = true .
20: eq empty =< S:Set = true .
21: eq (E:Elt S1:Set) =< S2:Set = (E in S2) and (S1 =< S2) .
22: cq (S1:Set = S2:Set) = (S1 =< S2) and (S2 =< S1) .
23: cq[es1s2]: (E:Elt in S1:Set) and (S1 =< S2:Set) = false if not(E in S2) .

24: --> generic sets with intersection operator
25: mod! SET^- (X :: TRIV) {
26: pr(SET=X))
27: op _^- : Set Set --> Set {assoc comm} .
28: eq empty ^ S2:Set = empty .
29: eq (E:Elt S1:Set) ^ S2:Set = if E in S2 then E (S1 ^ S2) else (S1 ^ S2) fi .
30: eq (S:Set ^ S) = S . }

31: --> module declaring proved lemmas on SET=^
32: mod! SETlem {
33: pr(SET=^-)
34: cq (A:Elt in (S1:Set S2:Set)) = (A in S1) or (A in S2)
35: if not(S1 == empty or S2 == empty) .
36: eq ((S1:Set =< S2:Set) and (S1 =< (A:Elt S2))) = (S1 =< S2) .
37: cq (S1:Set =< (A:Elt S2:Set)) = S1 =< S2 if not(A in S1) .
38: cq[es1s2]: (E:Elt in S1:Set) and (S1 =< S2:Set) = false if not(E in S2) .
39: cq[es1s2]: (E:Elt in S1:Set) and (S1 =< S2:Set) = false
40: if not(E in S2) . }

AID-QU (43:-45:) specifies queues of agent identifiers by renaming sorts of QUEhd (6.1.17:-22:). AID-SET (46:-51:) specifies sets of agent identifiers by renaming sorts and operator of SETlem (32:-40:), and adds the operator _-_as_. STATE (52:-59:) specifies the state space of QLOCK as the terms of the sort State that are constructed with a constructor [_r_w_c_], and adds the operator q->s_.

41: --> agent identifiers
42: mod* AID {[Aid]}
43: --> Queues of Aid (agent identifiers)
44: mod! AID-QU (X :: AID) {
45: pr(QUEhd(X{sort Elt -> Aid})*{sort Que -> Aq,sort EltEr -> AidEr})

AID-QU (43:-45:) specifies queues of agent identifiers by renaming sorts of QUEhd (6.1.17:-22:). AID-SET (46:-51:) specifies sets of agent identifiers by renaming sorts and operator of SETlem (32:-40:), and adds the operator _-_as_. STATE (52:-59:) specifies the state space of QLOCK as the terms of the sort State that are constructed with a constructor [_r_w_c_], and adds the operator q->s_.

41: --> agent identifiers
42: mod* AID {[Aid]}
43: --> Queues of Aid (agent identifiers)
44: mod! AID-QU (X :: AID) {
45: pr(QUEhd(X{sort Elt -> Aid})*{sort Que -> Aq,sort EltEr -> AidEr})
The modules WTtr (60:-65:), TYtr (66:-71:), EXtr (72:-78:) specify the want, try, exit behaviors of QLOCK (Section 6) as rewritings among terms of the sort State with transition rules [wt] (64:-65:), [ty] (70:-71:), [ex] (76:-78:) respectively. The transition rule [ex] can be without condition, but just for an example of conditional transition rules. The module QLOCK/TSP (80:) is just the module sum (i.e., conjunction) of the three modules WTtr, TYtr, EXtr, where “make name (modexp)” is equal to “mod name {pr(modexp)}”.

The majority of operators and equations in the module STATE is for constructing proof scores. The sort State’s reduction specification for QLOCK/TSP, which correspond to (Σ, E) of (Σ, E, Tl) (5.4.1), is the following STATE-b that is easily shown to be red-complete.

\[\text{STATE-b}\]

is in the file tsp-bare-state-spc.cafe on the Web.
6.2.2. Property Specifications for Mutual Exclusion Property of QLOCK/TSP

For proving the mutual exclusion property (MX property for short) of QLOCK/TSP, three modules INITprp (01:-10:), MXprp (11:-18:), HQ=Cprp (19:-24:) are prepared to define the three predicates init_ (07:-08:), mx_ (17:-18:), hq=c_ (22:-24:) respectively.

Example 8. (Simulation with Builtin Search Predicates)

CafeOBJ provides another builtin search predicate declared as follows.

| pred _=(*,*)=>*_suchThat_ : State State Bool |
|----------------------------------------------|
| eq s =(*,*)=>* SS:State suchThat p(SS)       |

3701:-24: is in the files tsp-init-prp.cafe, tsp-mx-inv-prp.cafe on the Web.
is reduced in a 3-tuple \((\Sigma, E, Tl)\), it behaves as follows. Note that the second argument is always a variable for binding found results. (i) Search for all the reachable states \(ss\) from \(s\) (i.e., \(s \rightarrow^*_{\Sigma,AC} ss\)), such that \(p(ss)\), (ii) if such states exist then print out all the found states and returns true, and (iii) if such states do not exist then returns false.

This builtin search predicate can be used to prove an invariant property with respect to finite transition systems. The following CafeOBJ code 25:-33, for example, proves that any QLOCK/TSP with up to 4 agents satisfies \(mx\) (mutual exclusion) property by checking the reductions 28:-29:, 30:-31:, 32:-33: return false. The parameter module “X.STATE :: AID” of the module (QLOCK/TSP + MXprp) is instantiated by the builtin module NAT with the view \{sort Aid \rightarrow Nat, \(op\ A1:Aid = A2:Aid \rightarrow A1:Nat == A2:Nat\}\} 26:-27:. The initial state is specified with the term like \([\text{nilQ r 1 2 3 w empS c empS}]\) (30:).

This builtin search predicate can also be used to find a counter example. Note that \([Q:Aq r ASr:As w ASw:As c ASc:As]\) denotes any term in \((T_{\Sigma})_{State}\). Assume we want to prove that \((\text{not } (ASr = \text{empS}) \text{ and } (ASc = \text{empS}))\) is an invariant. 34:-36: searches all reachable states from the initial state with 2 agents, and prints out two states \([ (2 | 1) r \text{empS w (2 1) c empS } ]\) and \([ (1 | 2) r \text{empS w (1 2) c empS } ]\) that satisfy \((\text{ASr = empS}) \text{ and } (\text{ASc = empS}))\) (i.e., two counter examples).

Simulation of finite transition systems with builtin search predicates, together with verifications of infinite systems with proof scores, is quite effective for specification analyses, verifications, and improvements (see Example 9).

6.2.3. Proof Scores for MX property of QLOCK/TSP

If \((mx S:State)\) is an invariant then the mutual exclusion property of QLOCK/TSP holds. \((mx S:State)\) is not an inductive invariant, and for using Fact 15 to prove \((mx S:State)\) is an invariant, \((hq=c S:State)\) is necessary to make \((mx S:State) \text{ and } (hq=c S))\) an inductive invariant.

28:25:-37: is in the files tsp-mx-simulation-bsp.cafe on the Web.
With the correspondences:

\[ \text{init}(S: \text{State}) \iff (\text{init } S: \text{State}) \text{ and } (\text{hq}=c \ S), \]

01:-20: is a proof score for INV1 and 21:-79: is a proof score for INV2 of Fact 15. Hence, 01:-79: is a proof score for proving that \((\text{mx } S: \text{State})\) is an invariant.

"st@ = \([q$ r sr$ w sw$ c sc$]\)" is the most general way to represent the fresh constant \(st@\) of the sort \(\text{State}\), and the case-split \(st\) (17:) is justified.

21: --> Current and Next state Relation with a data argument

23: --> module for defining check-cnr

24: mod! CHECKcnr (X :: CNR) \{ ... \}

25: -- predicate to check cnr against all 1 step transitions

21:-79: implements Fact 15 (INV2) by making use of Fact 18 with the following correspondences:

(1) \(\text{mx-iinv} \iff \text{iinv} (35:),\)
(2) \(\text{Aid} \iff \text{Data} (41:),\)
(3) \(\text{cnr-mx-iinv} \iff \text{cnr} (42:),\)
(4) \(\text{check-cnr-mx-iinv} \iff \text{check-cnr} (45:).\)

The definition of check-cnr, i.e., the equation just before Fact 18 is in the module CHECKcnr 23:-30:. Aid (Data) is not necessary and dropped from the arguments of check-cnr-mx-iinv (45:) by using dummy element dummyAid (44:).

21: --> defining mx initial condition

02: mod CHECK-mx-init \{
03: pr(INITprp + MXprp + HQ=Cprp)
04: pred check-mx-init_ : State .
05: eq check-mx-init S:State =
06: (init S) implies ((mx S) and (hq=c S)) .
07: -- fresh constant
08: op st@ : -> State .
09: -- goal proposition is "check-mx-init st@ = true"
10: -- fresh constants for refinements
11: ops a$ ar$ a$1 ac$1 ac$2 : -> Aid .
12: ops q$ q$1 : -> Aq .
13: ops sr$ sw$ sc$ sc$1 sc$2 : -> As . \}
14: --> executing proof of "check-mx-init st@ = true" with PTcalc
15: select CHECK-mx-init .
16: :goal(eq check-mx-init st@ = true .)
17: :def st = :csp(eq st@ = \([q$ r sr$ w sw$ c sc$]\) .)
18: :def q=nil# = :ctf(eq q$ = nilQ .)
19: :def sc=em# = :ctf(eq sc$ = empS .)
20: :apply(st q=nil# rd- sc=em# rd-)

20... (24:,30:) shows an abbreviation and the full description is in the file check-cnr.cafe on the Web.

\(\text{(24:30)}\)
pred check-cnr : State Data .
eq check-cnr(S:State,D:Data) =
not(S =(*,1)|| SS:State if CC:Bool
suchThat not((CC implies cnr(S,SS,D)) == true)
\{i(S,SS,D,CC) ... \} . }

module defining cnr-mx-iinv
mod CNR-mx-iinv {
pr(MXprp + HQ=Cprp)
pred mx-iinv : State .
eq mx-iinv(S:State) = ((mx S) and (hq=c S)) .
pred cnr-mx-iinv(S:State,SS:State,A:Aid) =
\{ (mx-iinv(S) implies mx-iinv(SS)) . \}

proposition to check whether cnr-mx-iinv holds
mod CHECK-cnr-mx-iinv {
inc(CHECKcnr(CNR-mx-iinv{sort Data -> Aid,
op cnr -> cnr-mx-iinv}))
pred check-cnr-mx-iinv : State .
op dummyAid : Aid .
eq check-cnr-mx-iinv(S:State) = check-cnr(S,dummyAid) .
-- fresh constant
op st@ : State .
-- goal proposition is: "check-cnr-mx-iinv(st@) = true"
-- fresh constants for refinements
ops a$ ar$ a$1 ac$1 ac$2 : Aid .
ops q$ q$1 : Aq .
ops sr$ sw$ sc$ sc$1 sc$2 : As . }

Fact \[18\] says that to show
\( \text{(CHECK-cnr-mx-iinv + QLOCK/TSP) \leq \text{check-cnr-mx-iinv(st@)} \) 
is sufficient for proving Fact \[15\] (INV2). QLOCK/TSP is \((WTtr + TYtr + EXtr) \) 
and
\( ((\text{CHECK-cnr-mx-iinv + WTtr}) \leq \text{check-cnr-mx-iinv(st@)} \) \)
\( ((\text{CHECK-cnr-mx-iinv + TYtr}) \leq \text{check-cnr-mx-iinv(st@)} \) \)
\( ((\text{CHECK-cnr-mx-iinv + EXtr}) \leq \text{check-cnr-mx-iinv(st@)} \) \)
\( \Rightarrow (\text{CHECK-cnr-mx-iinv + QLOCK/TSP}) \leq \text{check-cnr-mx-iinv(st@)} \).
\[53:-60; 61:-69; 70:-79: \] are three PTcalc proof scores for the above three premises. Hence, 21:-79: is a proof score for Fact \[15\] (INV2).

The case-split \wt\ (56:) refines \st\ into the most general instance with fresh constants of the left hand side of the transition rule \wt\ (6.2.1) in the module WTtr. Any transition with the transition rule \wt\ should happen from a term of sort State that is a refined instance of the left hand side, and the case-split \wt\ is justified. The same argument is valid for the case-splits \ty\ (64:) and \ex\ (73:).

executing proof mx-iinv \wt\nopen (CHECK-cnr-mx-iinv + WTtr) .
goal\{eq check-cnr-mx-iinv(st@) = true . \}
def \wt = :csp{eq st@ = [q$ x (ar$ sr$) w sv$ c sc$] .}
def q=nil = :csp{eq q$ = nilQ . eq q$ = (a$1 | q$1) .}
apply(wt q=nil rd-)
show proof
close
executing proof mx-iinv TY
open (CHECK-cnr-mx-iinv + TYtr).
:goal{eq check-cnr-mx-iinv(st@) = true .}
def ty = :csp{eq st@ = [(a$ | q$) r sr$ w (a$ sw$) c sc$] .}
def sc=em = :csp{eq sc$ = empS . eq sc$ = (ac$1 sc$1) .}
def a=ac1 = :ctf{eq a$ = ac$1 .}
apply(ty sc=em rd- a=ac1 rd-)
show proof
close

executing proof mx-iinv EX
open (CHECK-cnr-mx-iinv + EXtr).
:goal{eq check-cnr-mx-iinv(st@) = true .}
def ex = :csp{eq st@ = [(a$ | q$) r sr$ w sw$ c sc$] .}
def sc1=em = :csp{eq sc$1 = empS . eq sc$1 = (ac$2 sc$2) .}
def a=ac1 = :ctf{eq a$ = ac$1 .}
apply(ex sc=em rd- sc1=em rd- a=ac1 rd-)
show proof
close

Four “:show proof” commands for the four pieces of code 14:-20:; 53:-60:; 61:-69:; 70:-79: print out the following c01:-c24: that show the effectiveness of the proof trees constructed. (“:show proof” for 14:-20: is done after 20:.) A proof score
test for the module (CHECK-cnr-mx-iinv + QLOCK/TSP) is possible instead of 53:-79: but tends to be complex.

c01: root* c07: root* c11: root* c17: root*
c02: [st] 1* c08: [st] 1* c12: [cy] 1* c18: [ex] 1*
c03: [qnil#] 1-1* c09: [qnil] 1-1* c13: [sc] 1-1* c19: [sc] 1-1*
c04: [sc] 1-1-1* c10: [qnil] 1-2* c14: [sc] 1-2* c20: [sc] 1-2*
c05: [sc] 1-2-1* c15: [a] 1-2-1* c21: [sc] 1-2-1* c22: [a] 1-2-1*
c06: [qnil#] 1-2* c16: [a] 1-2-2* c23: [a] 1-2-2* c24: [sc] 1-2-2*

c08: [st] 1* c09: [st] 1* c13: [sc] 1-1* c19: [sc] 1-1*
c04: [sc] 1-1-1* c10: [qnil] 1-2* c14: [sc] 1-2* c20: [sc] 1-2*
c05: [sc] 1-2-1* c15: [a] 1-2-1* c21: [sc] 1-2-1* c22: [a] 1-2-1*
c06: [qnil#] 1-2* c16: [a] 1-2-2* c23: [a] 1-2-2* c24: [sc] 1-2-2*

Example 9. (Improving EXtr)
Suppose we want to change the EXtr’s transition rule 6.2.1 as follows.
\[ \text{tr}[\text{ex}]: [(A | Q) r Sr w Sw c (Ac Sc)] \Rightarrow [Q r (A Sr) w Sw c Sc]. \]
This rule does not require to check \( A = Ac \) and provides a more flexible protocol, and it is worthwhile to check whether it also satisfies the mx property. The proof can be achieved by adding \( ac\$ \) to 50: and replacing 73: as follows.

50: ops a$ ar$ a$1 ac$ ac$1 ac$2 : -> Aid .
73: :def ex = :csp(eq st@ = [(a$ | q$) r sr$ w sw$ c (ac$ sc$)].

The new rule \( \text{tr}[\text{ex}] \) is not conditional and the proof becomes simpler, and c22:-c23: are omitted.

6.2.4. Property Specifications for Leads-to Property of QLOCK/TSP
03:-05: defines two state predicates \(_\text{inw}_-, \_\text{inc}_-\) for defining a leads-to property \(_\text{inw}_- \text{leads-to} \_\text{inc}_-\) (WC property for short) which asserts that if an agent gets into \( ASw:As \) (Waiting section) then the agent will surely get into

\[\text{In the file tsp-mx-iinv-allrl-psc.cafe on the Web.}\]
ASc: As (Critical section).

02: --> defining (_inw_) (_inc_) for WC property
03: mod! WCprp { pr(STATE) preds (_inw_) (_inc_) : Aid State .
04: eq A:Aid inw [Q:Aq r ASr:As w ASw:As c ASc:As] = A in ASw .
05: eq A:Aid inc [Q:Aq r ASr:As w ASw:As c ASc:As] = A in ASc . }

For constructing proof scores for the WC property (_inw_ leads-to _inc_), the state predicates r^w, w^c, r^c, q=wc, qvr, qnd (08:-23:) need to be used as invariant properties inv(s) in Fact [16] (LT1) and (LT2).

06: --> defining invariant properties for WC property
07: mod! WCinvs { pr(STATE) preds (r^w_) (w^c_) (r^c_) (q=wc_) (qvr_) (qnd_) : State .
08: eq r^w [Q:Aq r ASr:As w ASw:As c ASc:As] = ((ASr ^ ASw) = empS) .
09: eq w^c [Q:Aq r ASr:As w ASw:As c ASc:As] = ((ASw ^ ASc) = empS) .
10: eq r^c [Q:Aq r ASr:As w ASw:As c ASc:As] = ((ASr ^ ASc) = empS) .
11: eq q=wc [Q:Aq r ASr:As w ASw:As c ASc:As] = ((Q->s Q) = (ASw ASc)) .
12: eq qvr [Q:Aq r ASr:As w ASw:As c ASc:As] = not(((Q->s Q) ASr) = empS) .
13: pred qnd_ : Aq .
14: eq qnd nilQ = true .
15: eq qnd (Q1:Aq | A:Aid | Q2:Aq) = not(A in ((Q->s Q1) (Q->s Q2))) and (qnd Q1) and (qnd Q2) .
16: eq qnd [Q:Aq r ASr:As w ASw:As c ASc:As] = qnd Q . }

6.2.5. Proof Scores for WC property of QLOCK/TSP

The proof scores for proving that the state predicates r^w, w^c, r^c, q=wc, qvr, qnd (08:-23:) are invariant properties of QLOCK/TSP can be constructed in a similar way as the proof scores (01:-79:) for the state predicates mx, hq=c, r^w, w^c, r^c, q=wc, qvr, qnd is invariant property, inv(s) in Fact [16] (LT1) and (LT2) can be specified with 03:-11:. As a matter of fact, 08; 11: are not necessary in the following proof scores.

01: --> module declaring inv_ for WC property of QLOCK/TSP
02: mod! INVlem { pr(MXprp + HQ=Cprp + WCinvs)
03: pred inv : State .
04: cq inv(S:State) = false if not(mx S) .
05: cq inv(S:State) = false if not(hq=c S) .
06: cq inv(S:State) = false if not(r^w S) .
07: cq inv(S:State) = false if not(w^c S) .
08: cq inv(S:State) = false if not(r^c S) .
09: cq inv(S:State) = false if not(q=wc S) .
10: cq inv(S:State) = false if not(qvr S) .
11: cq inv(S:State) = false if not(qnd S) .

3201:-23: is in the files tsp-wc-prp.cafe, tsp-wc-inv-prp.cafe on the Web.
33In the files tsp-wc-init-pac.cafe, tsp-wc-inv-pac.cafe on the Web.
3401:-87: is in the files tsp-wc-inv-lem.cafe, tsp-wc-dms-prp.cafe, tsp-wc-daq-lem.cafe, tsp-wc-wc1-psc.cafe, tsp-wc-wc2-psc.cafe on the Web.
indicates that implements Fact 16 (LT1) by making use of Fact 18 with the following correspondences:

1. inv (03) = inv,
2. _in w_ (6.2.4 03) = p,
3. _inc_ (6.2.4 03) = q,
4. #dms (24) = m,
5. Aid = Data (41),
6. cnr-wc1 = cnr (41),
7. check-cnr-wc1 = check-cnr (42).

15:26: defines #dms and its subsidiary operators _, #daq, _, and declares a lemma on #daq that is used at 51. PNAT*ac (14) specifies Peano natural numbers with _, _, _, and PNAT*ac> (33) specifies PNAT*ac plus _.

The PTcalc proof score on the modules (CHECK-cnr-wc1 + TYtr) and (CHECK-cnr-wc1 + EXtr) are constructed as the one on (CHECK-cnr-wc1 + WTtr) (49:60).

35:-38: The proof score for this lemma is in the file tsp-wc-daq-lem.cafe on the Web.
36: The specifications are in the file pnat-spc.cafe on the Web.
37: In the file tsp-wc-wc1-psc.cafe on the Web.
47: ops q$ q$1 : -> Aq .
48: ops sr$ sv$ sw$i sc$ sc$1 sc$2 : -> As .
49: --> executing proof of check-cnr-wc1-wt with PTcalc
50: open (CHECK-cnr-wc1 + WTtr) .
51: pr(DAQ-lem)
52: :goal{eq check-cnr-wc1(st@,aa@) = true .}
53: :def wt = :csp{eq st@ = [q$ r (ar$ sr$) w sv$ c sc$] .}
54: :def sc=em = :csp{eq sc$ = empS . eq sc$ = ac$1 sc$1 .}
55: :def aa%sw = :csp{eq sv$ = aa@ sv$1 . eq (aa@ in sv$) = false .}
56: :def aa=ar = :ctf{eq aa@ = ar$.}
57: :def aa!q = :ctf{eq (aa@ in (q->s q$)) = true .}
58: :apply(wt sc=em rd- aa%sw rd- aa=ar rd- aa!q rd-)
59: :show proof
60: close

65:--68: indicates that 61:68: implements Fact [LT2] by making use of Fact [10].

7. Discussion

7.1. Related Works

7.1.1. Specification languages

Many algebraic specification languages/systems have been developed, among them are OBJ [19], HISP [22], ASL [55], ASF+SDF [1], Larch [31], CafeOBJ [21], CASL, and Maude [35]. Some of them, including Larch, CASL, and Maude, have their own verification tools, and only OBJ and CafeOBJ adopt the proof score approach just with equational reduction engines. Maude is a sibling language of CafeOBJ and the two languages share many important features, but Maude’s Inductive Theorem Prover (MITP) [8] does not take the proof score approach.
Set theory based formal specification languages, like VDM [54], Z [58], B [3], are popular and widely used. The verification of specifications in these languages can be achieved with proof assistants like PVS [48], Isabelle/HOL [33], Rodin [52] by making use of stepwise refinements and seem to be practically effective. Proof scores with module reuses and refinements in CafeOBJ have a potential for realizing the proofs via stepwise refinements.

7.1.2. Theorem Provers

Interactive theorem provers or proof assistants have similar motivation as proof scores. Among the most noted proof assistants are ACL2 [1], Coq [9], Isabelle/HOL [33], PVS [48]. All of them are not algebraic and not based on equational deduction/reduction.

SAT/SMT solvers, e.g., MiniSAT [39], Z3 [50], and Yices [56], have a nice power to fully automate proofs in some specific classes, and are used in several interactive theorem provers. This kind of fully automated theorem provers can be used to discharge a sub-goal in a PTcalc proof tree. [49, 50] reported CiMPA/CiMPG/CiMPG-F for CafeInMaude (CiM) [51] that fully automated some proof scores for OTS’s invariant properties. CiMPA/CiMPG/CiMPG-F plus SAT/SMT have a potential to automate the PTcalc+WFI proof scores in a significant way.

7.1.3. CafeOBJ Related

The logical semantics of CafeOBJ is structured by the concept of institution [11]. Recent studies inheriting this tradition include [23, 26]. Recent focus on applications of specification verification with CafeOBJ is multitask, hybrid, real-time system. Recent publications in this category include [40, 42]. Proof scores for cloud protocols and automotive software standards still continue to be investigated. [57] reported an interesting rare achievement on proof scores for leads-to properties, and develops to domain specific reuse of proof scores.

CafeInMaude (CiM) [51] is the recent second implementation of CafeOBJ on the Maude system. Promising proof automation/assistant tools CiMPA/CiMPG/CiMPG-F have already been developed [49, 50] by making use of the Maude meta-programming facility.

7.2. Distinctive Features of Proof Scores

7.2.1. ADT and Model Based Proof

Comparing with the notable theorem provers like ACL2, Coq, Isabelle/HOL, PVS, the specification verification with proof scores in CafeOBJ has the following characteristics.

- Systems/services are modeled via Algebraic Abstract Data Types (ADT) with equations and an appropriate higher abstraction level can be settled for each system/property specification. Moreover, the equations are used directly as reduction/rewriting rules for proofs at the higher abstraction level.
The major two proof rules Fact 9 (PR1), Fact 11 (PR2) are simple and transparent, formalized at the level of specification satisfaction \( SP \models p \) (i.e., any model of \( SP \) satisfies \( p \)), and supporting model based proofs. Another important proof rule for inductions formalized as Fact 14 is also succinct and nicely harmonizes with the two rules (PR1), (PR2).

7.2.2. Reduction Based Proof

A proof score succeeds if each of the reductions involved returns the expected result, usually \textit{true}. Each reduction is with the equations in the current module. Hence, by observing the current equations and the failed reduction result, the equations to be added for getting desirable reduction result could be guessed. The following three items are all declared as equations that are going to be used in reductions, and have a good chance to be predicted through the current and expected reduction results. This is a nice and important feature based on the transparent reduction (rewriting) based proof.

- Necessary missing axioms and lemmas (see Example 6).
- Exhaustive equations for case-splits.
- Necessary induction hypotheses.

7.2.3. Initiality, Termination, Confluence, Sufficient Completeness

The properties proof scores prove are the semantic properties of interest on specifications. Each of the properties is supposed to be expressed in a CafeOBJ’s predicate (2.1.3), an invariant property (Fact 15), or a leads-to property (Fact 16). These properties are the primary indexes of the quality of specifications. That is, if a proof score succeeds in proving a property on a specification then the equations (i.e., axioms) in the specification are shown to be sufficient enough to imply the property.

Initiality (2.5), termination, confluence, and sufficient completeness (2.6.3) (ITCS for short) are another properties of specifications. They are also important indexes of the quality of specifications. Each specification is better to satisfy these properties as much as possible depending on each context. There are many studies on these properties, e.g., [28, 34, 38, 41, 53], that help to make a specification satisfy them.

\((\Sigma, E)\) part of a transition specification need to be red-complete for proof scores (with builtin search predicates) to be valid (5.4.1). For other kind of specifications, proof scores do not assume that each specification satisfies any of ITCS properties, but the followings are observed.

- A high quality specification with respect to ITCS properties has the high possibility of success in proofs with proof scores. Actually, a property relating to initial models could not be proved without some equations that are implied by the initiality. The last sentence of 3.1 explains that the equations \(3.1-11:-12: \) are of this kind.
- The lack of an ITCS property of a specification is sometimes detected in a middle of constructing proof scores on the specification. For example,
subtle non-terminating reduction rules (i.e., equations) in a specification could be detected by encountering a non-terminating proof score.

7.2.4. PTcalc+WFI Proof Scores

The case-split with equations in a PTcalc command :csp{...} has the following merits comparing to the one with the open ... close constructs.

- The exhaustive equations are declared intensively inside a :csp command and the user’s intention is clearly expressed in an easy to check style.
- Although CafeOBJ does not support yet, the exhaustiveness can be checked automatically in the majority of cases based on constructor declarations.

PTcalc is more fundamental than WFI and there are many PTcalc proof scores without WFI. However, term refinements (instances of Fact 11 (PR2), e.g., 4.2.3-19:-20:) are needed to do WFI, and WFI is always PTcalc+WFI. Many kinds of induction schemes including structural induction were coded into proof scores, and many cases were developed before PTcalc+WFI was prepared 12, 13. PTcalc+WFI subsumes all of these various induction schemes in a uniform and transparent way. This contributes to applicability and flexibility of PTcalc+WFI.

The conditions for PTcalc+WFI proof scores to be correct (sound) are intensively localized into the following three points.

- Exhaustiveness of the equations declared in a :csp{...} command.
- Validity of the equation declared in an :init command with (...), e.g., 4.1.2-10:
- Well-foundedness of the binary relation _wf>_ used to declare an induction hypothesis.

Although no automatic support from CafeOBJ yet, the well-foundedness of a binary relation _wf>_ on the argument tuples of a goal predicate can be established, in the majority of cases, via the strict subterm relations on constructor terms.

Full automation of PTcalc+WFI would be difficult because of at least the followings.

- A :csp command could be used to declare the interested class of models for which the equations in the command are exhaustive. The exhaustiveness can not be checked automatically in this case.
- It would be difficult to automate the procedure for finding (i) a predicate p and (ii) a well-founded relation on the argument tuples of p that lead to successful induction.

The transparent and flexible structure of PTcalc+WFI, however, could help to design effective interactive tools.
7.3. Conclusions

- A minimum of theories for justifying the practice of the proof score constructions in CafeOBJ is presented in a unified style, and important results are formulated in 19 Facts.
- Proof scores for case-split and/or induction are shown to be represented as the highly stylized proof scores with the proof tree calculus (PTcalc) and the well-founded induction (WFI).
- Concrete proof scores in the PTcalc+WFI style for non-trivial specifications of transition systems are demonstrated to be justified by the theories and the Facts.

[Acknowledgments] The author appreciates comments given by D. Găină, N. Hirokawa, M. Nakamura, K. Ogata, N. Preining, A. Riesco, T. Sawada, H. Yatsu, H. Yoshida.

Comments from anonymous referees were great help to improve the quality of the paper.

This work was supported in part by Grant-in-Aid for Scientific Research (S) 23220002 from Japan Society for the Promotion of Science (JSPS).

References

[1] ACL2, 2022.09 accessed. Web page. URL: https://www.cs.utexas.edu/users/moore/accl2/

[2] Astesiano, E., et al., 2002. CASL: the common algebraic specification language. Theor. Comput. Sci. 286, 153–196. doi:10.1016/S0304-3975(01)00368-1

[3] B, 2022.09 accessed. Web page. URL: https://formalmethods.fandom.com/wiki/B-Method

[4] van den Brand, M., et al., 2001. The Asf+Sdf meta-environment: a component-based language development environment. Electron. Notes Theor. Comput. Sci. 44, 3–8. doi:10.1016/S1571-0661(04)80917-4

[5] Burstall, R., 1969. Proving properties of programs by structural induction. Computer Journal 12(1), 41–48. doi:10.1093/comjnl/12.1.41

[6] CafeOBJ, 2022.09 accessed. Web page. URL: https://cafeobj.org/

[7] Chandy, K.M., Misra, J., 1989. Parallel program design - a foundation. Addison-Wesley.

[8] Clavel, M., Palomino, M., Riesco, A., 2006. Introducing the ITP tool: a tutorial. J. UCS 12, 1618–1650. doi:10.3217/jucs-012-11-1618

[9] Coq, 2022.09 accessed. Web page. URL: http://coq.inria.fr/
[10] Diaconescu, R., Futatsugi, K., 1998. CafeOBJ Report. volume 6 of AMAST Series in Computing. World Scientific. doi:10.1142/3831.

[11] Diaconescu, R., Futatsugi, K., 2002. Logical foundations of CafeOBJ. Theor. Comput. Sci. 285, 289–318. doi:10.1016/S0304-3975(01)00361-9.

[12] Futatsugi, K., 2006. Verifying specifications with proof scores in CafeOBJ, in: Proc. 21st IEEE/ACM ASE, IEEE. pp. 3–10. doi:10.1109/ASE.2006.73.

[13] Futatsugi, K., 2010. Fostering proof scores in CafeOBJ, in: Proc. 12th ICFEM (LNCS-6447), Springer. pp. 1–20. doi:10.1007/978-3-642-16901-4_1.

[14] Futatsugi, K., 2015. Generate & check method for verifying transition systems in CafeOBJ, in: Software, Services, and Systems (LNCS-8950), Springer. pp. 171–192. doi:10.1007/978-3-319-15545-6_13.

[15] Futatsugi, K., 2017. Introduction to Specification Verification in CafeOBJ (in Japanese). Saiensu-Sha, Tokyo. URL: https://cafeobj.org/iprog/.

[16] Futatsugi, K., 2020. Well-founded induction via term refinement in CafeOBJ, in: PreProc. WRLA 2020, pp. 64–78. URL: http://wrla2020.webs.upv.es/pre-proceedings.pdf.

[17] Futatsugi, K., 2021. Advances of proof scores in CafeOBJ: Invited paper, in: Intl. Symp. on Theoretical Aspects of Software Engineering TASE 2021, IEEE. pp. 3–12. doi:10.1109/TASE52547.2021.00012.

[18] Futatsugi, K., Găină, D., Ogata, K., 2012. Principles of proof scores in CafeOBJ. Theor. Comput. Sci. 464, 90–112. doi:10.1016/j.tcs.2012.07.041.

[19] Futatsugi, K., Goguen, J.A., Jouannaud, J.P., Meseguer, J., 1985. Principles of OBJ2, in: Proc. 12th ACM POPL (POPL85), ACM. pp. 52–66. doi:10.1145/318593.318610.

[20] Futatsugi, K., Goguen, J.A., Ogata, K., 2008. Verifying design with proof scores, in: Proc. VSTTE (LNCS-4171), Springer. pp. 277–290. doi:10.1007/978-3-540-69149-5.

[21] Futatsugi, K., Nakagawa, A.T., 1997. An overview of CAFE specification environment, in: Proc. First IEEE ICFEM, IEEE. pp. 170–182. doi:10.1109/ICFEM.1997.630424.

[22] Futatsugi, K., Okada, K., 1980. Specification writing as construction of hierarchically structured clusters of operators, in: Lavington, S.H. (Ed.), The 8th IFIP Congress 1980, North-Holland/IFIP. pp. 287–292.

[23] Găină, D., 2020. Forcing and calculi for hybrid logics. J. ACM 67, 25:1–25:55. doi:10.1145/3400294.
[38] Meseguer, J., 2017. Strict coherence of conditional rewriting modulo axioms. Theor. Comput. Sci. 672, 1–35. doi:10.1016/j.tcs.2016.12.026

[39] MiniSAT, 2022.09 accessed. Web page. URL: http://minisat.se

[40] Nakamura, M., Higashi, S., Sakakibara, K., Ogata, K., 2022. Specification and verification of multitask real-time systems using the OTS/CafeOBJ method. IEICE Trans. Fundam. Electron. Commun. Comput. Sci. 105-A, 823–832. doi:10.1587/transfun.2021map0007

[41] Nakamura, M., Ogata, K., Futatsugi, K., 2014. Incremental proofs of termination, confluence and sufficient completeness of OBJ specifications, in: Specification, Algebra, and Software (LNCS-8373), Springer. pp. 92–109. doi:10.1007/978-3-642-54624-2_5

[42] Nakamura, M., Sakakibara, K., Okura, Y., Ogata, K., 2021. Formal verification of multitask hybrid systems by the OTS/CafeOBJ method. Int. J. Softw. Eng. Knowl. Eng. 31, 1541–1559. doi:10.1142/S0218194021400118

[43] OBJ3, 2022.09 accessed. Web page. URL: https://www.kindsoftware.com/products/opensource/obj3/

[44] Ogata, K., Futatsugi, K., 2003. Proof scores in the OTS/CafeOBJ method, in: Proc. 6th IFIP WG 6.1 FMOODS 2003 (LNCS-2884), Springer. pp. 170–184. doi:10.1007/978-3-540-39958-2_12

[45] Ogata, K., Futatsugi, K., 2008. Proof score approach to verification of liveness properties. IEICE Trans. 91-D, 2804–2817. doi:10.1093/ietisy/e91-d.12.2804

[46] Preining, N., Ogata, K., Futatsugi, K., 2015. Liveness properties in CafeOBJ, in: Proc. 24th LOPSTR, 2014 (LNCS-8981), Springer. pp. 182–198. doi:10.1007/978-3-319-17822-6_11

[47] PTCalc Manual, 2022.09 accessed. Web page. URL: https://cafeobj.org/files/ptcalc.pdf

[48] PVS, 2022.09 accessed. Web page. URL: http://pvs.csl.sri.com/

[49] Riesco, A., Ogata, K., 2018. Prove it! inferring formal proof scripts from CafeOBJ proof scores. ACM Trans. Softw. Eng. Methodol. 27, 6:1–6:32. doi:10.1145/3208951

[50] Riesco, A., Ogata, K., 2022. An integrated tool set for verifying CafeOBJ specifications. J. Syst. Softw. 189, 111302. doi:10.1016/j.jss.2022.111302

[51] Riesco, A., Ogata, K., Futatsugi, K., 2017. A Maude environment for CafeOBJ. Formal Asp. Comput. 29, 309–334. doi:10.1007/s00165-016-0398-7

53
[52] Rodin, 2022.09 accessed. Web page. URL: \url{http://www.event-b.org}

[53] Terese, 2003. Term Rewriting Systems. Cambridge University Press.

[54] VDM, 2022.09 accessed. Web page. URL: \url{https://www.overturetool.org}

[55] Wirsing, M., 1986. Structured algebraic specifications: A kernel language. Theor. Comput. Sci. 42, 123–249. doi:\url{10.1016/0304-3975(86)90051-4}

[56] Yices, 2022.09 accessed. Web page. URL: \url{https://yices.csl.sri.com}

[57] Yoshida, H., Ogata, K., Futatsugi, K., 2015. Formalization and verification of declarative cloud orchestration, in: Proc. 17th ICFEM (LNCS-9407), Springer. pp. 33–49. doi:\url{10.1007/978-3-319-25423-4\_3}

[58] Z, 2022.09 accessed. Web page. URL: \url{https://formalmethods.fandom.com/wiki/Z_notation}

[59] Z3, 2022.09 accessed. Web page. URL: \url{https://github.com/Z3Prover}