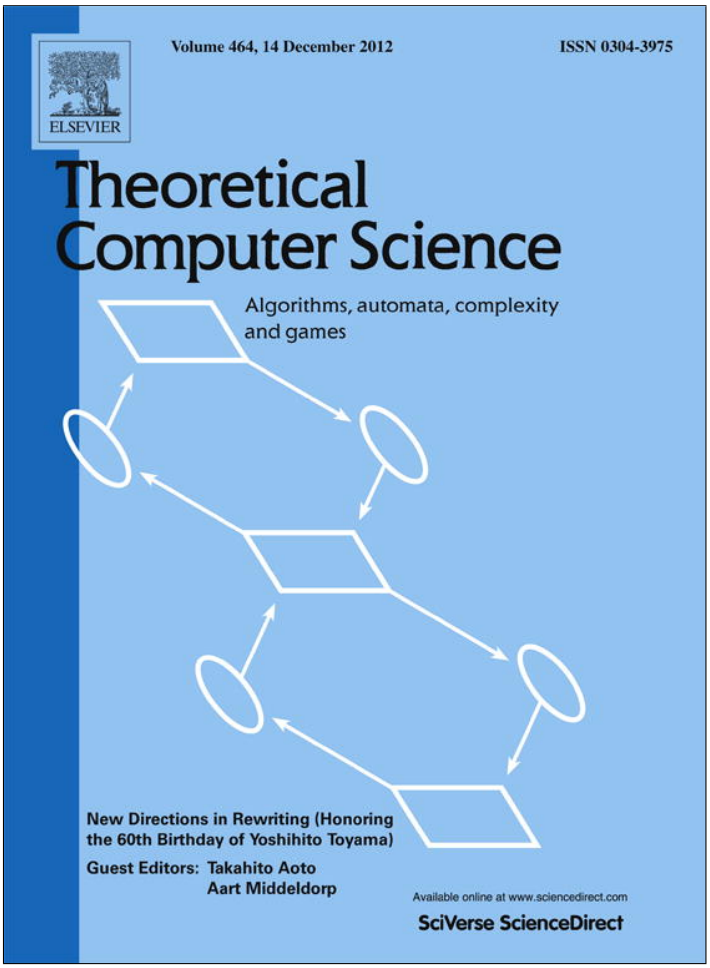


Theoretical Computer Science 464 (2012) 90–112

Contents lists available at SciVerse ScienceDirect

Theoretical Computer Science

journal homepage: www.elsevier.com/locate/tcs

Principles of proof scores in CafeOBJ

Kokichi Futatsugi *, Daniel Găină, Kazuhiro Ogata

Research Center for Software Verification & Graduate School of Information Science, Japan Advanced Institute of Science and Technology, 1-1 Asahidai, Nomi, Ishikawa 923-1292, Japan

ARTICLE INFO

Keywords:
Algebraic specifications
Theorem proving
Term rewriting
Proof scores
CafeOBJ

ABSTRACT

This paper describes the theoretical principles of a verification method with proof scores in the CafeOBJ algebraic specification language. The verification method focuses on specifications with conditional equations and realizes systematic theorem proving (or interactive verification). The method is explained using a simple but instructive example, and the necessary theoretical foundations, which justify every step of the verification, are described with precise mathematical definitions. Some important theorems that result from the definitions are also presented.

1. Introduction

Proof scores have been used by the OBJ algebraic specification language users from around 1985 to prove properties about OBJ specifications (or about the models the specifications define) [1–3].

Proof scores are instructions such that when executed (or "played"), if everything evaluates as expected, then the desired theorem is proved. A proof score is executed by applying proof measures, which progressively transform formulas in a language of goals into expressions that can be directly executed. Usually the proof measures are rewriting facilities that realize sound steps of equational reasoning.

Fully automatic systems verifications often fail to convey an understanding of their important structures, and they are generally unhelpful when they fail because of user errors in specifications or goals, or due to the lack of a necessary lemma or case splitting. It follows that one should seek to make 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. The tradeoff between detail and comprehension should also be carefully balanced. Proof scores are intended to meet these goals. Users of the CafeOBJ algebraic specification language [4–6] adopted the proof score verification method extensively and have fostered it into a powerful and systematic interactive verification method [7–13].

Verification methods with proof scores in CafeOBJ have been developed in conjunction with concrete proof efforts based on constructing specifications and proof scores for the areas of distributed algorithms/systems, real-time and/or hybrid systems, security protocols, e-commerce protocols, internal-controls for businesses, middleware protocols, etc. See [14,8, 12,13]. In parallel with these developments, formalization of the necessary logics and/or proof techniques has been done. The specification calculi in Section 6 formalize the basic mechanisms for the construction of proof scores based on these activities, and is expected to provide theoretical foundations for the future research developments on proof scores.

Similar ideas to the specification calculi are already presented in our papers [15,13] casually, but this paper presents our first serious formalization of the ideas with the following features.

- All the necessary concepts for formalizing the specification calculi are defined precisely and presented consistently in a uniform notation.

* Corresponding author. Tel.: +81 761 51 1255.
E-mail addresses: futatsugi@jaist.ac.jp (K. Futatsugi), daniel@jaist.ac.jp (D. Găină), ogata@jaist.ac.jp (K. Ogata).

0304-3975/$ – see front matter © 2012 Published by Elsevier B.V.
doi:10.1016/j.tcs.2012.07.041

- A simple but instructive example is included for motivating the formal definition of the specification calculi, and we try to make the paper as accessible as possible to motivated researchers/engineers who want to study and/or use formal methods with proper model theoretic semantics.
- The formalized specification calculi make our proof score method more precise and reliable. The calculi are intended to be used as a foundation for designing software tools for (1) checking the correctness of proof scores and (2) mechanically generating stylized parts of proof scores.

In the following sections, theoretical principles of the verification method with proof scores for constructor-based order-sorted equational specifications in CafeOBJ are described. Section 2 explains a simple but instructive example of specification and a proof score construction for verifying associativity of the append operation over the list data structure. It is intended to motivate subsequent developments. Section 3 defines needed basic mathematical concepts of signature, models, sentences, and satisfaction relations. Section 4 defines the equational calculi as basic proof measures (or proof engine) for the proof scores. Section 5 returns back to the motivating example of Section 2 and explains an example of proof score for case splitting. Section 6 defines the specification calculi for describing the constructions of proof scores formally. Section 7 concludes the paper with related works and future issues.

2. Motivating example: associativity of list append

The verification method with proof scores is explained using a simple example in a precise but semantically informal way. Important semantic concepts such as "model" and "satisfaction", which are used for explaining the example, will be defined formally in the following sections. The example used is the verification of associativity of the append operation on list data structure.

2.1. Constructing specification

Before doing any kind of verification, specification of a problem or system with respect to which the verification is going to be done should be defined.

Here is the specification of the list data structure in CafeOBJ. A line that starts with `--`, `-->`, `**`, or `**>` is a comment.

```
--> no automatic importation of built-in module BOOL
set include BOOL off
--> truth values of true and false
mod! TRUTH-VALUES{ [Bool]
  op true : -> Bool {constr}
  op false : -> Bool {constr}
}
--> trivial set of elements
mod* TRIV* {[Elt]}
--> parameterized list
mod* LIST (X :: TRIV*) {
  pr(TRUTH-VALUES)
  [Nil NnList < List]
  op nil : -> Nil {constr}
  op _|_ : Elt List -> NnList {constr}
  -- equality on the sort List
  op _=_ : List List -> Bool {comm}
  eq (L:List = L) = true .
  cq  L1:List = L2:List if (L1 = L2) .
}
```

"`set include BOOL off`" instructs the CafeOBJ system to stop automatic importation of the built-in module `BOOL`. This is for constructing the following specification without any presuppositions.

2.1.1. TRUTH-VALUES

`mod! TRUTH-VALUES{...}` defines a module with the name `TRUTH-VALUES` for defining the truth values `true` and `false` of the sort `Bool`. A set of elements of the same sort is denoted by a sort name, and the sort name is declared between `[` and `]`. "`op true : -> Bool {constr}`" declares that `true` is a constant (i.e. an operator without arguments) and is a constructor (`constr`) of the sort `Bool`. Another Boolean value of `false` is defined similarly. The CafeOBJ keyword `mod!` indicates that the module has the tight initial denotation, that is, this module denotes the initial model that consists of two distinct elements `true` and `false`, i.e., `Bool` = {`true`, `false`}.

2.1.2. TRIV*, LIST

The module TRIV* only declares the sort Elt. mod* is a keyword for declaring loose denotation, and indicates that the module denotes any model that satisfies the module. This module has nothing except the sort Elt and denotes any model that contains a (possibly empty) collection of entities of sort Elt.

As indicated by (X :: TRIV*), the module LIST has a parameter X that should satisfy the specification TRIV*, and defines a parameterized list structure. [Nil NnList < List] declares three sorts and sub-sort relations among them; sorts Nil and NnList are sub-sorts (i.e. are interpreted as subsets) of sort List. By declaring two operators nil and _|_ with constr (constructor) attribute, the sorts Nil, NnList, and List are defined to be constructed as follows.

$$\texttt{List} = \texttt{Nil} \cup \texttt{NnList}$$
$$\texttt{Nil} = \{\ \texttt{nil}\ \}$$
$$\texttt{NnList} = \{\ e \mid l \mid\ e \in \texttt{Elt},\ l \in \texttt{List}\}.$$

This implies that List can be represented as follows.

$$\texttt{List} = \{\texttt{nil},\ e_{00}\mid\texttt{nil},\ e_{10}\mid e_{11}\mid\texttt{nil},\ \ldots, e_{n0}\mid e_{n1}\mid\ldots e_{nn}\mid\texttt{nil},\ \ldots \mid e_{ij} \in \texttt{Elt},\ i,j \in \{0,1,2,\ldots\}\}.$$

The module LIST imports the module TRUTH-VALUES with the protecting (pr) mode; this means that any model of LIST includes the model of TRUTH-VALUES without introducing any new element and any new equality. Notice that the module LIST is declared with loose denotation (i.e. mod*) and denotes the class of models that satisfy the module.

2.1.3. Equation and equality predicate

A binary predicate (operator that returns a value of the sort Bool) _=_ over the sort List is defined, and is declared to be commutative by the comm attribute. The keyword eq indicates the declaration of an equation. The CafeOBJ equation

```
eq (L:List = L) = true .
```

represents a universally quantified equation

$$(\forall L : \texttt{List})\ \ (L = L) = \texttt{true}$$

and declares that any two elements of List are equal if they are denoted by two expressions (i.e. terms) with the same normal form. See Sections 2.2.1, 6.4 for the meaning of "normal form". Notice that in a CafeOBJ equation a declaration like L:List of a variable and its sort appears at the place where the variable is used first in an equation, and the variable is effective until the end of the equation. The keyword cq indicates the declaration of a conditional equation, and declares that an equation holds if the condition declared after if holds.

Notice also that the eq's first and the cq's second equality symbols represent the equality predicate. Whereas, the cq's first and the eq's second equality symbols are parts of cq and eq of the CafeOBJ language.

It is worthwhile to notice that the two equations in the module LIST (i.e. eq and cq) guarantee the logical equivalence of the CafeOBJ language level (i.e. meta level) equality and sort level (i.e. object level) equality.[1]

2.1.4. APPEND, APPEND-ASSOC

The append operation over lists and its associativity are defined by the following two modules of APPEND and APPEND-ASSOC.

```
--> append _@_ operation on List
mod* APPEND(X :: TRIV*){
  pr(LIST(X))
  -- append operation on List
  op _@_ : List List -> List
  eq nil @ L2:List = L2 .
  eq (E:Elt | L1:List) @ L2:List = E | (L1 @ L2) .
}
--> associativity of _@_ (append)
mod* APPEND-ASSOC(X :: TRIV*){
  pr(APPEND(X))
  -- "_@_" is associative
  op @assoc : List List List -> Bool
  eq @assoc(L1:List,L2:List,L3:List)
     = ((L1 @ L2) @ L3 = L1 @ (L2 @ L3)) .
}
```

pr(LIST(X)) in the module APPEND declares that the module imports LIST(X) with the protecting mode, where X in LIST(X) is the parameter X :: TRIV*.

[1] "cq L1:List = L2:List if (L1 = L2) ." is not executable as a left to right rewriting rule in CafeOBJ, because the left hand of the equation is a variable.

2.2. Constructing proof scores for verifying associativity

Verification of associativity of the append operation with respect to the specification APPEND-ASSOC is formalized as "verifying that any model of APPEND-ASSOC satisfies the equation $((\forall L1, L2, L3 : \texttt{List})\texttt{@assoc}(L1, L2, L3) = \texttt{true})$". This is written as follows.

$$\texttt{APPEND-ASSOC} \models ((\forall L1, L2, L3 : \texttt{List})\ \texttt{@assoc}(L1, L2, L3) = \texttt{true}).$$

This can also be written as follows by using the CafeOBJ variable declaration notation and an abbreviation $SP \models p$ for $SP \models (p = \texttt{true})$, where SP is a specification and p is a predicate.

```
APPEND-ASSOC |= @assoc(L1:List,L2:List,L3:List)     (SA-AA).
```

An assertion like $SP \models p$ means that any model of SP satisfies p. A proof score is the CafeOBJ code (i.e. text in the CafeOBJ language) for verifying this **satisfaction assertion**.

2.2.1. A possible initial proof score

The following is a possible proof score for verifying the associativity of the list append, that is, for verifying the above satisfaction assertion (SA-AA).

```
-- check whether "@assoc(L1:List,L2:List,L3:List)"
-- is deducible at "APPEND-ASSOC"
--> [0] the goal
red in APPEND-ASSOC : @assoc(L1:List,L2:List,L3:List) .
--> returns "(((L1 @ L2) @ L3) = (L1 @ (L2 @ L3)))"
```

"red in APPEND-ASSOC : @assoc(L1:List,L2:List,L3:List) ." is a command of CafeOBJ to get a **normal form** of the term @assoc(L1:List,L2: List,L3:List) by using all the equations of APPEND-ASSOC as left to right rewriting rules as much as possible.

Here L1:List, L2:List, L3:List are declarations of fresh constants that are effective only inside the reduction command. This means that the CafeOBJ system creates a temporal module, say AA-TEMP, (1) with loose denotation that (2) protects APPEND-ASSOC and (3) declares fresh constants "ops L1 L2 L3 : -> List", and get the normal form of @assoc(L1,L2,L3) by reduction at AA-TEMP. (1), (2), and (3) imply that any model M' of the module AA-TEMP is just a model M of APPEND-ASSOC plus an assignment of three elements of the sort List of M to the three constants L1, L2, L3. This means that "@assoc(L1,L2,L3) is true for any model M' of AA-TEMP" implies "@assoc(L1:List,L2:List,L3:List) is true for any model M of APPEND-ASSOC", and vice versa. Hence, (SA-AA) holds if and only if AA-TEMP $\models$ @assoc(L1,L2, L3). This is exactly the Theorem of Constants, a proof rule that is defined in Section 6.2.2.

As it is well known, reduction is partial but correct simulation of equational deduction. Therefore, if the reduction command returns true, then APPEND-ASSOC $\models$ @assoc(L1:List,L2:List,L3:List). Hence, the reduction command is a possible proof score for the satisfaction assertion (SA-AA).

2.2.2. Induction induced by constructors

Unfortunately, the above reduction command does not return true, and it is not a proof score. It is also well known that the inductive structure of the sort List needs to be used for proving associativity of append. The inductive structure of the sort List is defined in the module LIST by declaring the two operators nil and _|_ with the constructor (constr) attribute. These constructor declarations intend to restrict the class of models of LIST to the reachable [2] ones. A model M of the module LIST interprets the sort Elt as a set $M_{\texttt{Elt}}$, the sort Nil as a set $M_{\texttt{Nil}}$, the sort NnList as a set $M_{\texttt{NnList}}$, the sort List as a set $M_{\texttt{List}}$, the operator nil as a function $M_{\texttt{nil}} : \texttt{->} M_{\texttt{Nil}}$, and the operator _|_ as a function $\_M_{|}\_ : M_{\texttt{Elt}}\ M_{\texttt{List}} \texttt{->} M_{\texttt{NnList}}$. A model M of LIST is defined to be **reachable** if $M_{\texttt{List}}$ is represented as follows.

$$M_{\texttt{List}} = \{\, M_{\texttt{nil}},\ e_{00}M_{|}M_{\texttt{nil}},\ e_{10}M_{|}e_{11}M_{|}M_{\texttt{nil}},\ \ldots,\ e_{n0}M_{|}e_{n1}M_{|}\ldots e_{nn}M_{|}M_{\texttt{nil}},\ \ldots \mid e_{ij} \in M_{\texttt{Elt}}, i, j \in \{0, 1, 2, \ldots\}\,\}.$$

That is, any element of $M_{\texttt{List}}$ can be constructed with $M_{\texttt{Elt}}$, $M_{\texttt{nil}}$, and $\_M_{|}\_$. Because of the constructor declarations of the operators nil and _|_, any model of the module LIST is defined to be a reachable one. This makes it possible to use structural induction with respect to the sort List.

[2] We call model "reachable" iff (if and only if) it is "generated" by constructors, that is, any element of the model is equal to the denotation of a constructor generated term.

2.2.3. A proof score for induction and its proof rule

The following is a proof score for the satisfaction assertion `(SA-AA)`.

```
**> decide to use structural induction w.r.t.
**> the first argument l1 of "@assoc(L1:List,L2:List,L3:List)"
--> Induction base
--  check whether "@assoc(nil,L2:List,L3:List)" is deducible
--  at "APPEND-ASSOC"
--> [00] sub-goal 0 for the goal [0]
red in APPEND-ASSOC : @assoc(nil,L2:List,L3:List) .
--> returns "true"
--> Induction step
mod* APPEND-ASSOC-iStep(X :: TRIV*){
pr(APPEND-ASSOC(X))
-- arbitrary element l1:List
  op l1 : -> List
-- induction hypothesis "@assoc(l1,L2:List,L3:List) = true"
  eq (l1 @ L2:List) @ L3:List = l1 @ (L2 @ L3) .
-- arbitrary element e:Elt
  op e : -> Elt }
--  check whether "@assoc(e | l1,L2:List,L3:List)" is deducible
--  at "APPEND-ASSOC-iStep"
--> [01] sub-goal 1 for the goal [0]
red in APPEND-ASSOC-iStep : @assoc(e | l1,L2:List,L3:List) .
--> returns "true"
--> QED
```

If the reduction command "`red in APPEND-ASSOC : @assoc(nil,L2:List, L3:List) .`" returns `true`, it implies the following.

$$\texttt{APPEND-ASSOC} \models \texttt{@assoc(nil,L2:List,L3:List)} \qquad \texttt{(SA-AA-0)}.$$

The module `APPEND-ASSOC-iStep` is for checking the induction step. It includes a declaration of a constant "`op l1 :-> List`" for an arbitrary element of the sort `List`, an equation for declaring the induction hypothesis, and a declaration of a constant "`op e :-> Elt`" for an arbitrary element of the sort `Elt`. If the reduction command "`red in APPEND-ASSOC-iStep :@assoc(e | l1,L2:List,L3:List) .`" returns `true`, it implies that `@assoc(e | l1,L2:List,L3:List)` is equationally deducible in `APPEND-ASSOC-iStep`. Hence the following.[3]

$$\texttt{APPEND-ASSOC-iStep} \models \texttt{@assoc(e | l1,L2:List,L3:List))} \qquad \texttt{(SA-AA-1)}.$$

The two satisfaction assertions `(SA-AA-0)` and `(SA-AA-1)` constitute a sufficient condition for the satisfaction assertion `(SA-AA)`, if the models of the module `APPEND-ASSOC` are restricted to the reachable ones. This implication can be written in a standard style of proof rule as follows. Notice that this is not a proof rule at the syntactic level.

$$\frac{\texttt{APPEND-ASSOC} \models \texttt{@assoc(nil,L2:List,L3:List)} \qquad \texttt{APPEND-ASSOC-iStep} \models \texttt{@assoc(e | l1,L2:List,L3:List))}}{\texttt{APPEND-ASSOC} \models \texttt{@assoc(L1:List,L2:List,L3:List)}}.$$

The two reduction commands in the above CafeOBJ code return `true` and it has turned out to be a proof score.[4] It implies that the satisfaction assertion `(SA-AA)` is verified and the associativity of the list append is proved.

3. Models and satisfaction

CafeOBJ supports several kinds of specifications such as equational specifications, rewriting specifications, and behavioral specifications, but this paper focuses on equational specifications, for which our proof score method has been mainly developed.

For defining models and satisfaction relations with respect to the class of constructor-based order-sorted equational specifications, the following concepts will be defined.[5]

---

[3] Notice that this satisfaction assertion is equivalent to the following.

$$\texttt{APPEND-ASSOC} \models (\forall e : \texttt{Elt})(\forall l1 : \texttt{List})(((\forall L2, L3 : \texttt{List})\texttt{@assoc}(l1, L2, L3)) \Rightarrow ((\forall L2, L3 : \texttt{List})\texttt{@assoc}(e \mid l1, L2, L3))).$$

[4] The reader is encouraged to download the CafeOBJ system from http://www.ldl.jaist.ac.jp/cafeobj/system.html and feed the above CafeOBJ code to the system and check that the two reduction commands return `true`.

[5] We are taking an approach that is influenced by the institution [16] following the style of [4]. But we do not use the institution in this paper.

- a class $\mathbb{S}ign$ of **signatures**,
- for each signature $\Sigma \in \mathbb{S}ign$ a class $\textsc{Mod}(\Sigma)$ of Σ-**models**,
- for each signature Σ a set $Sen(\Sigma)$ of Σ-**sentences**, and
- for each signature Σ a **satisfaction relation** $\models_\Sigma$ between Σ-models and Σ-sentences.

A specification SP is in practice a finite collection of sentences E for the same signature Σ, and is formally defined as a pair consisting of a signature Σ and the collection of sentences in $Sen(\Sigma)$. That is, $SP = (\Sigma, E)$. If $SP = (\Sigma, E)$, Σ is denoted by $\mathbb{S}ig(SP)$.

The **denotation** of a specification is the class of all the models (i.e. possible implementations) that satisfy the specification. It represents the semantics, or the meaning, of the specification. In fact, this is one of the characteristic features of algebraic specifications: a specification is just a formal description of a certain class of models, which exist only ideally as an abstract mathematical object.

A specification that is defined by giving a signature Σ and a set of sentences E directly is called a basic specification. A specification that is defined by using other already defined specifications is called a structured specification. We first define the denotations of basic specifications; the denotations of structured specifications are then defined in Section 3.5.

A specification can have a loose denotation or a tight denotation. The **loose denotation** of a specification is the class $\textsc{Mod}(SP)$ of all models of $\mathbb{S}ig(SP)$ that satisfy all sentences in SP. The **tight denotation** consists only of the **initial model** 0_{SP} in $\textsc{Mod}(SP)$, i.e., for any model $M \in \textsc{Mod}(SP)$ of SP there exists a unique model morphism $0_{SP} \to M$. Initial models are unique up to isomorphisms, so essentially there is only one initial model. Hence, by "the" initial model we often mean the whole (isomorphism) class of initial models. CafeOBJ supports the distinction between loose and tight denotations by the special keywords, `mod*` for loose semantics, and `mod!`for tight semantics.

We will define $\textsc{Mod}(SP)$ and 0_{SP} in the following.

3.1. Signatures

Signatures are formed by a poset (partially order set) of sorts and operators typed using that poset of sorts.

3.1.1. Sorts

A **sort** is a name for entities of the same type. Sorts constitute the basis for the CafeOBJ type system. Semantically, a sort denotes the set of entities of that type (sort). CafeOBJ supports subtyping via a subsort declaration that specifies an inclusion between two sets. If the subsort declaration of `s < s'` is given, this means that the set of elements of sort `s` is a subset of the set of elements of sort `s'`. Notice that `s1 s2 < s` is an abbreviation for "`s1 < s` and `s2 < s`", etc. With the subsort declarations, the set of sorts S is understood as the poset $(S, \leq)$ where $\leq$ is the smallest partial order including all the ordered pairs declared with <.

Given a poset of sorts $(S, \leq)$, let $\equiv_\leq$ denote the equivalence relation generated by the partial order $\leq$. The quotient of S under the equivalence relation $\equiv_\leq$ is denoted by $\hat{S} = S/\equiv_\leq$, and an element of $\hat{S}$ is called a **connected component** of the ordered set of sorts $(S, \leq)$. An element of $\hat{S}$ that contains a sort s is denoted by $[s]$.

3.1.2. Operators

An **operator** (or function) f on a set of sorts S is denoted by $f : w \to s$ where $w \in S^*$ is its **arity** and $s \in S$ is its **sort** (sometimes called **co-arity**). The string ws is called the **rank** of the operator. **Constants** are operations whose arity is empty, i.e., $f : [] \to s$.

Let F_{ws} denotes the set of all operations of rank ws, then the whole collection of operators F can be represented as the family of sets of operators sorted by (or indexed by) ranks as $F = \{F_{ws}\}_{w \in S^*, s \in S}$. Notice that $f : w \to s$ iff $f \in F_{ws}$.

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 $w's'$.

CafeOBJ has a built-in module `BOOL` with the sort `Bool`, and an operator with co-arity `Bool` is called **predicate**.

3.1.3. Order-sorted signature

An **order-sorted signature** is defined by a tuple $(S, \leq, F)$. For making possible the construction of symbolic presentations of models (i.e. term algebras) of a signature, the following condition of **sensibility** is a most general sufficient condition for avoiding ambiguity found until now [17].

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 name $f \in F_{ws} \cap F_{w's'}$. Where $w \equiv_\leq w'$ means that (1) w and w' are of the same length, say, n, and (2) for $1 \leq i \leq n$ the i-th element of w is in the same connected component as that of the i-th element of w'. Notice that $[] \equiv_\leq []$ for the empty arity []. An order-sorted signature is always assumed to be sensible in this paper.

Example 1. In the CafeOBJ notation, `{[Bool Nat] op 0 :-> Bool op 0 :-> Nat}` defines a non-sensible signature, and `0` cannot be identified with any entity of any sort, while, `{[Zero < Nat EvenInt] op 2 :-> Nat op 2 :-> EvenInt}` defines a sensible signature and `2` is identified with an entity that belongs to both `Nat` and `EvenInt`, but it has no minimal parse. □

3.1.4. Constructor-based order-sorted signature

A **constructor-based order-sorted signature** is an order-sorted signature with constructor declarations and is represented by a tuple $(S, \leq, F, F^c)$, where $(S, \leq, F)$ is an order-sorted signature, and $F^c \subseteq F$ is a distinguished subfamily of sets of operators, called **constructors**. That is, $F^c = \{F^c_{ws}\}_{w\in S^*, s\in S}$ and $F^c_{ws} \subseteq F_{ws}$. It is also assumed that $(S, \leq, F^c)$ is an order-sorted signature and, since $(S, \leq, F)$ is sensible, $(S, \leq, F^c)$ is also sensible.[6]

A sort $s \in S$ is called **constrained** if

1. there exists an operator $f \in F^c_{ws}$ with the co-arity s, or
2. there exists a constrained sort s' such that $s' \leq s$.

Let S^c denote the set of all constrained sorts. A sort that is not constrained is called **loose**, and the set of loose sorts is denoted by S^l. That is, $S^l = S - S^c$.

Example 2. The module `LIST` of Section 2.1 determines the constructor-based order-sorted signature $Sig(\texttt{LIST}) = (S, \leq, F, F^c)$ as follows. $S = \{\texttt{Bool}, \texttt{Elt}, \texttt{Nil}, \texttt{NnList}, \texttt{List}\}$. $< = \{(\texttt{Nil List}), (\texttt{NnList List})\}$. $F = \{F_{ws}\}_{w\in S^*, s\in S}$ where $F_{\texttt{Bool}} = \{\texttt{true}, \texttt{false}\}$, $F_{\texttt{Nil}} = \{\texttt{nil}\}$, $F_{\texttt{Elt List NnList}} = \{\_\,|\,\_\}$, $F_{\texttt{List List Bool}} = \{\_=\_\}$, $F_{ws} = \{\}$ otherwise. $F^c = \{F^c_{ws}\}_{w\in S^*, s\in S}$ where $F^c{}_{\texttt{Bool}} = \{\texttt{true}, \texttt{false}\}$, $F^c{}_{\texttt{Nil}} = \{\texttt{nil}\}$, $F^c{}_{\texttt{Elt List NnList}} = \{\_\,|\,\_\}$, $F^c{}_{ws} = \{\}$ otherwise. $S^c = \{\texttt{Bool}, \texttt{Nil}, \texttt{NnList}, \texttt{List}\}$. $S^l = \{\texttt{Elt}\}$. □

3.2. Models

Models lie at the heart of the semantics of algebraic specification.

3.2.1. $(S, \leq, F)$-algebras

Let $(S, \leq, F)$ be an order-sorted signature. An $(S, \leq, F)$-**algebra** (or an order-sorted algebra of signature $(S, \leq, F)$) M interprets

- each sort $s \in S$ as a set M_s,
- each subsort relation $s < s'$ as an inclusion $M_s \subseteq M_{s'}$, and
- each operator $f \in F_{s_1\ldots s_n s}$ as a function $M_f : M_{s_1} \times \cdots \times M_{s_n} \to M_s$ such that any two operators of the same name return the same value if applied to the same argument, i.e. if $f : w \to s$ and $f : w' \to s'$ and $ws \equiv_{\leq} w's'$ and $\overline{a} \in M_w \cap M_{w'}$ then $M_{f:w\to s}(\overline{a}) = M_{f:w'\to s'}(\overline{a})$.

That is, the $(S, \leq, F)$-**algebra** M consists of the following.

(A1) The order-sorted family of carrier sets $\{M_s\}_{s\in S}$ satisfying $(s \leq s' \Rightarrow M_s \subseteq M_{s'})$, and
(A2) the set of functions

$$\{M_f : M_{s_1} \times \cdots \times M_{s_n} \to M_s \mid f \in F_{s_1\ldots s_n s}, F = \{F_{ws}\}_{w\in S^*, s\in S}\}$$

such that any two functions with the same name return the same value when applied to the same arguments.

3.2.2. $(S, \leq, F)$-algebra-morphism

Let M and N be $(S, \leq, F)$-algebra. An $(S, \leq, F)$-**algebra-morphism** (or model-morphism) $h : M \to N$ is an S-sorted family of functions between the carriers of M and N, $\{h_s : M_s \to N_s\}_{s\in S}$, such that

(M1) $h_s(M_f(a_1, \ldots, a_n)) = N_f(h_{s_1}(a_1), \ldots, h_{s_n}(a_n))$ for all $f \in F_{s_1\ldots s_n s}$, and $a_i \in M_{s_i}$ for $i \in \overline{1..n}$, and
(M2) if $s \equiv_{\leq} s'$ and $a \in M_s \cap M_{s'}$ then $h_s(a) = h_{s'}(a)$.

Notice that we use the notation $\overline{i..j}$ for denoting the set of successive natural numbers $\{i, \ldots, j\}$.

If there exist algebra-morphisms $h : M \to N$ and $g : N \to M$ such that $h; g : M \to M$ is the identity map on M, h is called **isomorphism**. In this case, M and N are called isomorphic.

3.2.3. 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 set of variables. The notion of $\Sigma(X)$-**term** is defined inductively as follows. Notice that sensibility makes the definition consistent.

- 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 $f \in F_{s_1\ldots s_n s}$ and terms t_i of sort s_i for $i \in \overline{1..n}$.

6 In general, if $F' \subseteq F$ then $((S, \leq, F)$ is sensible) implies $((S, \leq, F')$ is sensible).

The S-sorted set of $\Sigma(X)$-terms is denoted as $T_\Sigma(X) \stackrel{\text{def}}{=} \{T_\Sigma(X)_s\}_{s\in S}$. For the S-sorted empty sets of variables $\{\}$, a $\Sigma(\{\})$-term is called Σ-term (or **ground**-term). $T_\Sigma \stackrel{\text{def}}{=} T_\Sigma(\{\})$ denotes the S-sorted set of Σ-ground-terms.

Both $T_\Sigma(X)$ and T_Σ can be organized as Σ-algebras in the obvious way by using the above inductive definition of Σ-terms. CafeOBJ is a language for modeling systems in Σ-algebras.

T_Σ has the following **initiality** property [3,18].

Fact 3. *Let $\Sigma = (S, \leq, F)$ be an order-sorted signature. For any Σ-algebra M there exists a unique Σ-algebra-morphism $T_\Sigma \to M$.* □

3.2.4. $(S, \leq, F, F^c)$-algebras

An $(S, \leq, F, F^c)$-**algebra** (or a constructor-based order-sorted algebra of signature $(S, \leq, F, F^c)$) M is an $(S, \leq, F)$-algebra where the carrier sets for the constrained sorts consist of interpretations of terms formed with constructors and elements of loose sorts. That is, the following holds for $\Sigma^c = (S, \leq, F^c)$.

(A3) There exists an S^l-sorted sets of loose variables Y $(= \{Y_s\}_{s\in S^l})$, and an S^l-sorted function $\theta : Y \to M \stackrel{\text{def}}{=} \{\theta_s : Y_s \to M_s\}_{s\in S^l}$ such that for every constrained sort $s \in S^c$ the function $\theta^\#_s : (T_{\Sigma^c}(Y))_s \to M_s$ is a surjection, where $\theta^\#$ is the unique extension of θ to an Σ^c-algebra-morphism (see Section 3.4.1).

3.3. Sentences: equations

Sentences of equational specifications are equations. Let $\Sigma = (S, \leq, F)$ be an order-sorted signature, and $t \in (T_\Sigma(X))_s$, $t' \in (T_\Sigma(X))_{s'}$ for some S-sorted set of variables X and sorts s, s' that belong to the same connected component (i.e. $s, s' \in [s'']$ for some sort s''). An equational atom is defined to be a pair of terms (t, t'), which is written as $t = t'$. A conditional Σ-**equation** is defined as

$$(\forall X)\ t = t'\ \textbf{if}\ \{t_1 = t'_1, \ldots, t_n = t'_n\}$$

where X is an S-sorted set of variables, $t = t'$ is an equational atom, and $\{t_1 = t'_1, \ldots, t_n = t'_n\}$ is a set of equational atoms called the **condition** of the equation.[7] When the condition is empty it is called an **unconditional** equation, and is written as

$$(\forall X)\ t = t'.$$

3.4. Satisfaction

The satisfaction relation between models and sentences is at the heart of the semantics of algebraic specifications.

3.4.1. Valuations and term interpretation

Valuations assign values to variables. In other words, they instantiate the variables with values from a given model. Let Σ be an $(S, \leq, F)$-signature. Given a Σ-model M and an S-sorted set X of variables, a valuation $v : X \to M$ consists of an S-sorted family of maps $\{v_s : X_s \to M_s\}_{s\in S}$.

Each Σ(X)-term t can be interpreted as a value $v(t)$ in the model M for each valuation $v : X \to M$ in the following inductive manner:

- M_f if t is a constant f,
- $v(x)$ if t is a variable x,
- $M_f(v(t_1), \ldots, v(t_n))$ if t is of the form $f(t_1, \ldots, t_n)$ for some $f \in F_{s_1\ldots s_n s}$ and terms t_i of sort s_i.

3.4.2. Equational satisfaction, $M \models_\Sigma E$, $E \models_\Sigma E_0$

Let Σ be a signature. A Σ-equation $(\forall X)\ t = t'$ **if** $\{t_1 = t'_1 \ldots, t_n = t'_n\}$ is **satisfied** by a Σ-algebra M, denoted as

$$M \models_\Sigma ((\forall X)\ t = t'\ \textbf{if}\ \{t_1 = t'_1 \ldots, t_n = t'_n\})$$

iff $v(t) = v(t')$ holds whenever $((\forall i \in \overline{1..n})\ v(t_i) = v(t'_i))$ holds for any valuation $v : X \to M$. For any set E of Σ-equations, $M \models_\Sigma E$ if for all $e \in E$, we have $M \models_\Sigma e$. A set E of Σ-equations satisfies a set E_0 of Σ-equations, $E \models_\Sigma E_0$, when for all Σ-models M, we have $M \models_\Sigma E$ implies $M \models_\Sigma E_0$.

---

[7] In the CafeOBJ language definition given by Diaconescu and Futatsugi [4], $\{t_1 = t'_1, \ldots, t_n = t'_n\}$ is defined to be a term of the sort `Bool`. The reason why we adopt the more general definition in this paper is that we are using examples that do not assume the special module `BOOL` or the special sort `Bool` in that module. The two definitions are interchangeable by the transformations $p \to \{p = \texttt{true}\}$ and $\{t_1 = t'_1, \ldots, t_n = t'_n\} \to (t_1 = t'_1\ \textit{and}\ \ldots\ \textit{and}\ t_n = t'_n)$.

3.4.3. Signature inclusions and reducts

We say that $\Sigma_0 = (S_0, \leq_0, F_0, F_0^c)$ is **included** in $\Sigma = (S, \leq, F, F^c)$, in symbols $\Sigma_0 \subseteq \Sigma$, when

1. $S_0 \subseteq S$, for all $w_0 \in S_0^*$ and $s_0 \in S_0$, we have $(F_0)_{w_0 s_0} \subseteq F_{w_0 s_0}$ and $(F_0^c)_{w_0 s_0} \subseteq F^c_{w_0 s_0}$, and
2. no "new" constructors are introduced by F^c for "old" constrained sorts, i.e. if $s_0 \in S_0^c$ and $\sigma \in F^c_{ws_0}$ then $w \in S_0^*$ and $\sigma \in (F_0^c)_{ws_0}$.

By Gâinâ [15], the Σ_0-part of any Σ-model M, denoted by $M\restriction_{\Sigma_0}$, is reachable by the constructors in F_0^c, i.e. $M\restriction_{\Sigma_0} \in \text{Mod}(\Sigma_0)$. $M\restriction_{\Sigma_0}$ is called a **reduct** of M through the inclusion $\Sigma_0 \subseteq \Sigma$. By Goguen and Meseguer [18], the following satisfaction condition holds for all Σ-models M and Σ_0-equations ε.

$$M \models_\Sigma \varepsilon \quad \text{iff} \quad M\restriction_{\Sigma_0} \models_{\Sigma_0} \varepsilon.$$

3.4.4. Substitutions and reducts

Given a signature $\Sigma = (S, \leq, F, F^c)$ and a set of non-constructor constants X, disjoint from Σ, we let $\Sigma(X)$ to denote $(S, \leq, F \cup X, F^c)$. The models of $\Sigma(X)$ can be viewed as pairs (M, f), where M is a Σ-model and $f : X \to M$ is a valuation. The model M is called the **reduct** of (M, f) to the signature Σ, in symbols $(M, f)\restriction_\Sigma = M$.

A **substitution** is a function $\theta : X \to T_\Sigma(Y)$, where Σ is a signature, X and Y are sets of non-constructor constants disjoint from Σ. The reduct of a $\Sigma(Y)$-model (M, f) along the substitution θ is $(M, \theta; f^\#)$, in symbols $(M, f)\restriction_\theta = (M, \theta; f^\#)$, where $f^\# : T_\Sigma(Y) \to M$ is the unique extension of f to a Σ-morphism.

By Gâinâ [15], for every $\Sigma(Y)$-model M, we have $M\restriction_\theta \in \text{Mod}(\Sigma(X))$. By Goguen and Meseguer [18], the following satisfaction condition holds for all $\Sigma(X)$-equations ε, and $\Sigma(Y)$-models M.

$$M\restriction_\theta \models_{\Sigma(X)} \varepsilon \text{ iff } M \models_{\Sigma(Y)} \theta(\varepsilon).$$

3.5. Structured specifications

We consider the following four specification building operations **BS**, **SU**, **PR**, and **IN** for constructing a new specification from old ones. The specification building operations presented here were introduced in [19].

(**BS**) A specification SP is built by giving its signature and set of equations. That is, $SP = (\Sigma, E)$ and
- $\text{Sig}(SP) \stackrel{\text{def}}{=} \Sigma$,
- $\text{Mod}(SP) \stackrel{\text{def}}{=} \text{Mod}(\Sigma, E)$.

(**SU**) A new specification $SP_1 + SP_2$ is built by making sum of two specifications SP_1 and SP_2 with the same signature Σ.
- $\text{Sig}(SP_1 + SP_2) \stackrel{\text{def}}{=} \text{Sig}(SP_1) = \text{Sig}(SP_2) = \Sigma$,
- $\text{Mod}(SP_1 + SP_2) \stackrel{\text{def}}{=} \text{Mod}(SP_1) \cap \text{Mod}(SP_2)$.

(**PR**) Given a specification SP and a signature Σ such that $\text{Sig}(SP) \subseteq \Sigma$, the translation $\text{PR}(SP, \Sigma)$ of SP to Σ is defined as follows:
- $\text{Sig}(\text{PR}(SP, \Sigma)) \stackrel{\text{def}}{=} \Sigma$,
- $\text{Mod}(\text{PR}(SP, \Sigma)) \stackrel{\text{def}}{=} \{M \in \text{Mod}(\Sigma) \mid M\restriction_{\text{Sig}(SP)} \in \text{Mod}(SP)\}$.

(**IN**) Consider the specifications SP and SP_0 such that $\text{Sig}(SP_0) = \Sigma_0$, $\text{Sig}(SP) = \Sigma$, $\Sigma_0 \subseteq \Sigma$, and for all models $M \in \text{Mod}(SP)$ we have $M\restriction_{\Sigma_0} \in \text{Mod}(SP_0)$.[8] The free restriction $SP\,!_{SP_0}$ of SP modulo SP_0 is defined as follows:
- $\text{Sig}(SP\,!_{SP_0}) \stackrel{\text{def}}{=} \Sigma$,
- $\text{Mod}(SP\,!_{SP_0}) \stackrel{\text{def}}{=} \{M \in \text{Mod}(SP) \mid \text{for all } M\restriction_{\Sigma_0} \stackrel{h_0}{\to} N\restriction_{\Sigma_0} \text{ with } N \in \text{Mod}(SP) \text{ there is a unique } M \stackrel{h}{\to} N \text{ s.t. } h\restriction_{\Sigma_0} = h_0\}$.

Here $M \stackrel{h}{\to} N$ is a model-morphism $h : M \to N$ (see Section 3.2.2), and $h\restriction_{\Sigma_0}$ denotes the morphism h restricted to Σ_0. When $SP_0 = \emptyset$ then we simply write $SP\,!$. Notice that $\text{Mod}(SP\,!) = \{M \in \text{Mod}(SP) \mid \text{for all } N \in \text{Mod}(SP)$ there exists a unique $M \stackrel{h}{\to} N\}$, which is the initial model of $\text{Mod}(SP)$ if it exists.

Semantics of usual construction of a new module, by adding signature and equations to a protected existing module, can be defined as a successive application of **PR**, **SU** and/or **IN**. We denote by *SPEC* the class of all structured specifications.

Definition 4. A specification SP satisfies a set E of $\text{Sig}(SP)$-equations, in symbols $SP \models E$, when for all models $M \in \text{Mod}(SP)$, we have $M \models_{\text{Sig}(SP)} E$. □

[8] In this case SP is a refinement of SP_0, i.e. $SP_0 \subseteq SP$. See Section 3.5.2.

3.5.1. Flattening structured specifications

Any structured specification SP can be **flattened** to a basic specification $Flat(SP)$ as follows:

- $Flat(\Sigma, E) = (\Sigma, E)$ for any basic specification (Σ, E),
- $Flat(SP_1 + SP_2) = (\Sigma, E_1 \cup E_2)$ when $Flat(SP_i) = (\Sigma, E_i)$,
- $Flat(\mathrm{PR}(SP, \Sigma')) = (\Sigma', E)$ when $Sig(SP) \subseteq \Sigma'$ and $Flat(SP) = (\Sigma, E)$, and
- $Flat(SP\ !_{SP_0}) = Flat(SP)$.

Notice that the four specification building operations **BS**, **SU**, **PR**, and **IN** in Section 3.5 are defined via models, but the operation *Flat* is not.

3.5.2. Refinements

We say that SP is a **refinement** of SP_0, and denote it by $SP_0 \subseteq SP$, when

1. $Sig(SP_0) \subseteq Sig(SP)$, and for all models $M \in \mathrm{Mod}(SP)$, we have $M|_{Sig(SP_0)} \in \mathrm{Mod}(SP_0)$, or
2. $Sig(SP_0) = \Sigma(X)$, $Sig(SP) = \Sigma(Y)$, where Σ is a signature and X, Y are sets of non-constructor constants disjoint from the symbols in Σ, and there exists $\theta : X \to T_\Sigma(Y)$ such that for all $\Sigma(Y)$-models M, $M|_\theta \in Mod(SP_0)$.

Fact 5 ([20]). *Any specification SP refines its flattened specification $Flat(SP)$ (i.e. $Flat(SP) \subseteq SP$). And if SP is formed only from* **BS**, **SU** *and* **PR** *then* $\mathrm{Mod}(SP) = \mathrm{Mod}(Flat(SP))$. □

3.6. Initial algebras

Let $\Sigma = (S, \leq, F)$ be an order-sorted signature, which is sensible by definition in this paper, and let M be a Σ-algebra. Let $[s] \in S/\equiv_\leq$ be the connected component of sorts to which s belongs, and $M_{[s]} \stackrel{\text{def}}{=} \cup_{s' \in [s]} M_{s'}$. A Σ-**congruence** $\equiv$ on an order-sorted Σ-algebra M is a family of equivalence relations $\{\equiv_{[s]}\}_{[s] \in S/\equiv_\leq}$ over $\{M_{[s]}\}_{[s] \in S/\equiv_\leq}$ that satisfy the following condition.[9]

(C) For each $f \in F_{s_1 \ldots s_n s} \cap F_{s'_1 \ldots s'_n s'}$ and $a_i \in M_{s_i}$, $a'_i \in M_{s'_i}$ for $i \in \overline{1..n}$, if $a_i \equiv_{[s_i]} a'_i$ for $i \in \overline{1..n}$ then

$$M_{f:s_1,\ldots,s_n \to s}(a_1, \ldots, a_n) \equiv_{[s]} M_{f:s'_1,\ldots,s'_n \to s'}(a'_1, \ldots, a'_n).$$

Given an order-sorted Σ-algebra M and a Σ-congruence $\equiv$ on M, the quotient Σ-algebra $M/\equiv$ is defined as follows.

(Q1) For each $s \in S$,

$$(M/\equiv)_s \stackrel{\text{def}}{=} \{[a] \in (M_{[s]}/\equiv_{[s]}) \mid \exists a' \in M_{s'} \text{ such that } a \in [a']\}.$$

(Q2) For each $f \in F_{s_1 \ldots s_n s}$ and $[a_i] \in (M/\equiv)_s$ $(i \in \overline{1..n})$,

$$(M/\equiv)_{f:s_1,\ldots,s_n \to s}([a_1], \ldots, [a_n]) \stackrel{\text{def}}{=} [M_{f:s_1,\ldots,s_n \to s}(a'_1, \ldots, a'_n)]$$

where $a'_i \in [a_i] \cap M_{s_i}$ $(i \in \overline{1..n})$. Notice that this definition is independent in choice of representatives $a'_i (i \in \overline{1..n})$ because $\equiv$ is a congruence on M.

Notice the following.

- If $s \leq s'$ then $(M/\equiv)_s \subseteq (M/\equiv)_{s'}$ because $M_s \subseteq M_{s'}$.
- If $f \in F_{ws} \cap F_{w's'}$ and $ws \equiv_\leq w's'$ and $\overline{[a]} \in (M/\equiv)_{ws} \cap (M\equiv)_{w's'}$ then $(M/\equiv)_{f:w \to s}(\overline{[a]}) = (M/\equiv)_{f:w' \to s'}(\overline{[a]})$ because $\equiv$ is a congruence on M.

3.6.1. The initial algebra of order-sorted algebras

Given a set E of equations for order-sorted signature of $\Sigma = (S, \leq, F)$, then we can construct the Σ-algebra $T_{\Sigma,E}$ as follows.

- Consider the term Σ-algebra T_Σ (see Section 3.2.3), and define a family of equivalence relations $\equiv^E = \{\equiv^E_{[s]}\}_{[s] \in S/\equiv_\leq}$ over $\{(T_\Sigma)_{[s]}\}_{[s] \in S/\equiv_\leq}$ with

$$(t \equiv^E_{[s]} t') \stackrel{\text{def}}{=} ((\Sigma, E) \models (\forall\{\})\, t = t').$$

Notice $((\Sigma, E) \models (\forall\{\})\, t = t')$ is an instance of $SP \models E$ (see the Definition 4, Section 3.5). Notice also that $\equiv^E$ is a congruence on T_Σ because the equational deduction is reflective, symmetric, transitive, and congruent (see Section 4).

9 The definition of Σ-congruence and the constructions of quotient algebra and initial term algebra are suggested by José Meseguer through personal communication and his Lecture Note 11 for CS 467 (Program Verification).

- Define $T_{\Sigma,E}$ as the quotient Σ-algebra $T/\equiv^E$. Notice that for each $f \in F_{s_1\ldots s_n s}$ and $t_i \in (T_\Sigma)_{s_i}$ ($i \in \overline{1..n}$),

$$(T_{\Sigma,E})_{f:s_1,\ldots,s_n\to s}([t_1],\ldots,[t_n]) = [f(t_1,\ldots,t_n)].$$

The Σ-algebra $T_{\Sigma,E}$ has the following **initiality** property (see [3,18]), and is the model giving the tight denotation of the equational specification (Σ, E).

Fact 6. *Let (Σ, E) be an equational specification of order-sorted signature Σ, then for any Σ-algebra M satisfying all equations in E, there exists a unique Σ-algebra-morphism $T_{\Sigma,E} \to M$.* □

Notice that, in general, Fact 6 does not hold if Σ is extended to contain constructors (i.e. Σ is a constructor-based order-sorted signature). See Corollary 9.

3.6.2. The initial algebra of constructor-based order-sorted algebras

Let SP be a constructor-based order-sorted specification with the signature $\Sigma = (S, \leq, F, F^c), S^c$ be the set of constrained sorts, S^l be the set of loose sorts, $F^{S^c} \stackrel{\text{def}}{=} \{f : w \to s \mid f \in F, s \in S^c\}$, $\Sigma^{S^c} \stackrel{\text{def}}{=} (S, \leq, F^{S^c})$, $\Sigma^c \stackrel{\text{def}}{=} (S, \leq, F^c)$, and Y be any S^l-sorted set of variables.

Definition 7. A specification SP is **sufficiently complete** if for any term $t \in T_{\Sigma^{S^c}}(Y)$, where Y is a finite set of variables of loose sorts, there exists a term $t' \in T_{\Sigma^c}(Y)$ such that $(\Sigma^{S^c}, E) \models (\forall Y)t = t'$, where $Flat(SP) = (\Sigma, E)$. □

Notice that any specification without constructors is sufficiently complete by definition. Sufficient completeness is a sufficient condition for the existence of the initial algebra of constructor-based order-sorted algebras.

Theorem 8 (*[21]*). *Let $SP_0 \subseteq SP$ be a refinement such that $SP_0 = (\Sigma_0, E_0)$, $SP = (\Sigma, E)$, and SP is sufficiently complete. Then*

$$\mathrm{Mod}(SP\,!_{SP_0}) = \{M \in \mathrm{Mod}(SP) \mid \textit{there is } T_{\Sigma,E}(M|_{\Sigma_0}) \xrightarrow{h} M \textit{ isomorphism}\}.$$

Notice that, if $\Sigma_0 = (S_0, \leq_0, F_0, F_0^c)$, $M|_{\Sigma_0}$ can be seen as S_0-sorted set of constants $\{M_{s_0}\}_{s_0 \in S_0}$. □

Corollary 9. *If the specification (Σ, E) is sufficiently complete then for any Σ-algebra M satisfying all the equations in E, there exists a unique morphism $T_{\Sigma,E} \to M$.* □

4. Equational calculi

The equational calculi[10] for deducing equations from an initial set of equations are well established [18,22]. We are formalizing the equational calculi as entailment systems in this section. A signature Σ considered in this section is a constructor-based order-sorted signature unless otherwise stated.

4.1. The basic equational calculus

Definition 10. An **equational entailment system** [23] for deducing new equations from a set of equations consists of a family of entailment relations

$$\_\vdash\_ \stackrel{\text{def}}{=} \{\_\vdash_\Sigma\_\}_{\Sigma\in|\mathbb{S}ign|}$$

between sets of equations that satisfies the following **equational entailment rules**.

$$\text{[axiom-e]}\ \frac{E_0 \subseteq E}{E \vdash_\Sigma E_0} \qquad \text{[union-e]}\ \frac{E \vdash_\Sigma E_1 \quad E \vdash_\Sigma E_2}{E \vdash_\Sigma E_1 \cup E_2}$$

$$\text{[lemma-e]}\ \frac{E \cup E_0 \vdash_\Sigma E_1 \quad E \vdash E_0}{E \vdash_\Sigma E_1}$$

$$\text{[trans1-e]}\ \frac{E \vdash_\Sigma E_0 \quad \Sigma \subseteq \Sigma'}{E \vdash_{\Sigma'} E_0} \qquad \text{[trans2-e]}\ \frac{E \vdash_{\Sigma(X)} E_0 \quad \theta : X \to \Sigma(Y)}{\theta(E) \vdash_{\Sigma(Y)} \theta(E_0)}.$$

Here E, E_0, E_1, E_2 are sets of equations. □

An example of equational entailment system is given by the satisfaction relation $\_\models\_ = \{\_\models_\Sigma\_\}_{\Sigma\in|\mathbb{S}ign|}$. Notice that $E \models E'$ denotes that any model that satisfies E satisfies E'. See Section 3.4.2.

10 The terminology "calculus" is used to indicate, in contrast with semantics, the machinery of formally processing syntactic entities.

Definition 11. The **basic equational calculus** is defined as the least entailment system that satisfies (or is closed to) the equational entailment rules (i.e. [axiom-e], [union-e], [lemma-e], [trans1-e], and [trans2-e]) and the following proof rules that are called **the equational rules**.

$$[\text{reflexivity}]\ \emptyset \vdash_\Sigma (\forall X)\ t = t \qquad [\text{symmetry}]\ (\forall X)\ t = t' \vdash_\Sigma (\forall X)\ t' = t$$
$$[\text{transitivity}]\{(\forall X)\ t = t',\ (\forall X)\ t' = t''\} \vdash_\Sigma (\forall X)\ t = t''$$
$$[\text{congruence}]\{(\forall X)\ t_i = t_i' \mid i \in \overline{1..n}\} \vdash_\Sigma (\forall X) f(t_1, \ldots, t_n) = f(t_1', \ldots, t_n').$$

Here X is S-sorted set of variables, $t, t', t'' \in T_\Sigma(X)$, $f \in F_{s_1 \ldots s_n s}$, $t_i \in T_\Sigma(X)_{s_i}$ and $t_i' \in T_\Sigma(X)_{s_i'}$ for $i \in \overline{1..n}$.

$$[\text{instantiation}]\ \begin{array}{l}\{(\forall Y)\ t = t' \quad \textbf{if}\ \{t_i = t_i' \mid i \in \overline{1..n}\}\} \cup \\ \{(\forall X)\ \theta(t_i) = \theta(t_i') \mid i \in \overline{1..n}\} \vdash_\Sigma (\forall X)\ \theta(t) = \theta(t').\end{array}$$

Here $\theta : Y \to T_\Sigma(X)$ is a substitution.

Notice that this calculus is for deducing an unconditional equation, and an equation is considered a singleton set. □

A syntactic entity like $E \vdash_\Sigma E'$ is called an equational entailment or an $(E \vdash_\Sigma E')$-type entailment. A calculus that is composed of rules only with equational entailments is called an **equational calculus**. Notice that because an equational calculus indicates a signature Σ explicitly in an equational entailment $E \vdash_\Sigma E'$, it is for the basic or flattened specifications and not for the structured specifications.

Proposition 12. *For any signature Σ, the basic equational calculus is sound, i.e. $E \vdash_\Sigma E_0$ implies $E \models_\Sigma E_0$ for all E, $E_0 \subseteq \mathbb{S}en(\Sigma)$.*

Proof Sketch. The proof is the same as in [18]. □

Proposition 13 (*[18]*)**.** *The basic equational calculus is complete, i.e. $E \models_\Sigma E_0$ implies $E \vdash_\Sigma E_0$ for all E, $E_0 \subseteq \mathbb{S}en(\Sigma)$, if Σ has no constructors.* □

Proposition 13 will be extended for the signature with constructors in Theorem 16.

4.2. The constructor-based equational calculus

The basic equational calculus in Section 4.1 is not sufficient because many interesting properties are proved only for reachable models or the initial model of the specification (Σ, E).[11]

This section develops an equational calculus for the signature with constructors that enjoys the completeness theorem under the condition of the sufficient completeness.

Definition 14 (*[15]*)**.** The **constructor-based equational calculus** is defined as the least entailment system that includes the basic equational calculus, and satisfies the following rules that are called **the equational constructor rules**.

$$[\text{imp1-e}]\ \frac{E \vdash_\Sigma t = t' \text{ if } \{t_1 = t_1', \ldots, t_n = t_n'\}}{E \cup \{t_1 = t_1', \ldots, t_n = t_n'\} \vdash_\Sigma t = t'}$$
$$[\text{imp2-e}]\ \frac{E \cup \{t_1 = t_1', \ldots, t_n = t_n'\} \vdash_\Sigma t = t'}{E \vdash_\Sigma t = t' \text{ if } \{t_1 = t_1', \ldots, t_n = t_n'\}}.$$

Here E is any set of Σ-equations, and t, t', t_i, t_i' are ground terms. The rules [imp1-e], [imp2-e] show that equations in the condition of a conditional equation are interchangeable with equations in the premise of the entailment $\_\vdash_\Sigma\_$.

$$[\text{thConst1-e}]\ \frac{E \vdash_\Sigma (\forall Y)\varepsilon}{E \vdash_{\Sigma(Y)} \varepsilon} \qquad [\text{thConst2-e}]\ \frac{E \vdash_{\Sigma(Y)} \varepsilon}{E \vdash_\Sigma (\forall Y)\varepsilon}.$$

Here E is any set of Σ-equations, $(\forall Y)\varepsilon$ is any Σ-equation, and $\Sigma(Y)$ is the extension of Σ with constants from Y. The rules [thConst1-e] and [thConst2-e] show that variables in an equation are interchangeable with fresh constants (i.e. constants without name clashes).

$$[\text{subst-e}]\ (\forall X)\varepsilon \vdash_\Sigma (\forall Y)\theta(\varepsilon)$$

Here $(\forall X)\varepsilon$ is a Σ-equation, and $\theta : X \to T_\Sigma(Y)$ is a substitution.

$$[\text{abst-e}]\ \frac{\{\ E \vdash_\Sigma (\forall Y)\theta(\varepsilon) \mid \theta : X \to T_{\Sigma^c}(Y),\ Y\text{-finite set of loose variables}\ \}}{E \vdash_\Sigma (\forall X)\varepsilon}.$$

Here E is any set of Σ-equations, $(\forall X)\varepsilon$ is a Σ-equation, and Σ^c is the sub-signature of constructors. This rule [abst-e] says that if ε holds for all the possible constructor-based instantiations of variables in X, then $(\forall X)\ \varepsilon$ holds. □

11 The initial model, if it exists, is a reachable model.

The authors have proved in [15] that the constructor-based equational calculus is sound, and complete modulo sufficient completeness.

Proposition 15 (*Soundness [15]*)**.** *The constructor-based equational calculus is sound, i.e. for any sets E and E_0 of Σ-equations, we have $E \vdash_\Sigma E_0$ implies $E \models_\Sigma E_0$.* □

Theorem 16 (*Quasi Completeness Theorem [15]*)**.** *The constructor-based equational calculus is complete with respect to the sufficiently complete specifications, i.e. for any sets E and E_0 of Σ-equations such that (Σ, E) is sufficiently complete, we have $E \models_\Sigma E_0$ implies $E \vdash_\Sigma E_0$.* □

In Section 6, we will lift up the above proof rules to the level of structured specifications such that soundness is preserved (for all specifications), while completeness holds only for the sufficiently complete specifications build with the operations **BS**, **SU** and **PR**.

5. The motivating example again

Another important technique for constructing proof scores is case splitting. This section gives a simple example of case splitting using an enhanced version of `APPEND-ASSOC`.

5.1. Enhanced specification

The specification `APPEND-ASSOC` is enhanced to `APPENDvo-ASSOC` by introducing a void element `vo` of sort `Elt`. And the definition of the append operator `_@_` is changed to erase void element `vo` if it appears as the top of element `e` of the first argument of `_@_`. By this change, the verification of associativity needs to split the case into the two cases `(e = vo)` and `not(e = vo)`.

```
-- using built-in BOOL
set include BOOL on
--> a set of elements with a void element
mod* TRIVvo {[Elt] us(EQL) op vo : -> Elt}
--> parameterized list
mod* LISTvo (X :: TRIVvo){
  [Nil NnList < List]
  op nil : -> Nil {constr}
  op _|_ : Elt List -> NnList {constr} }
--> append _@_ operation on lists with a void element
mod* APPENDvo(X :: TRIVvo){
  pr(LISTvo(X))
  -- append operation on List with a void element
  op _@_ : List List -> List .
  eq [@1]: nil @ L2:List = L2 .
  eq [@2]: (E:Elt.X | L1:List) @ L2:List
            = if (E = vo) then (L1 @ L2)
              else E | (L1 @ L2) fi . }
--> associative predicate about _@_
mod* APPENDvo-ASSOC(X :: TRIVvo){
  pr(APPENDvo(X))
  -- "_@_" is associative
  pred @assoc : List List List .
  eq @assoc(L1:List,L2:List,L3:List)
     = ((L1 @ L2) @ L3 = L1 @ (L2 @ L3)) . }
```

The "`set include BOOL on`" command makes the automatic importation of the built-in module `BOOL` effective. The module `BOOL` imports the another built-in module `TRUTH-VALUE` and the truth values of sort `Bool` (i.e.`true` and `false`) are available. It also makes `if_then_else_fi` operator available. `us(EQL)` in the module `TRIV*` makes automatic declarations of the equality predicate `_=_` (see Section 2.1.3) effective for every sort.[12] In this case, it is assumed that the equality is declared as follows for every sort `*St*`.

```
op _=_ : *St* *St* -> Bool {comm}
eq (E:*St* = E) = true .
cq  E1:*St* = E2:*St* if (E1 = E2) .
```

Notice that `vo` is declared to be erased in `eq [@2]` of the module `APPENDvo`.

12 In the latest version, the declaration of `us(EQL)` is not necessary if the command "`set include BOOL on`" is used.

5.2. An enhanced proof score

A proof score for verifying the satisfaction assertion (SA-AA) (see Section 2.2) for APPENDvo-APPEND can be constructed in the same manner as previously until the induction step case as follows.

```
-- check whether "@assoc(L1:List,L2:List,L3:List)"
-- is deducible at "APPENDvo-ASSOC"
--> [0] the goal
red in APPENDvo-ASSOC : @assoc(L1:List,L2:List,L3:List) .
--> returns "(((L1 @ L2) @ L3) = (L1 @ (L2 @ L3)))"
**> decide to use structural induction w.r.t.
**> the first argument l1 of "@assoc(L1:List,L2:List,L3:List)"
--> Induction base
--  check whether "@assoc(nil,L2:List,L3:List)" is deducible
--  at "APPENDvo-ASSOC"
--> [00] sub-goal 0 for the goal [0]
red in APPENDvo-ASSOC : @assoc(nil,L2:List,L3:List) .
--> returns "true"
--> Induction step
mod* APPENDvo-ASSOC-iStep(X :: TRIVvo){
pr(APPENDvo-ASSOC(X))
-- arbitrary element l1:List
  op l1 : -> List
-- induction hypothesis "@assoc(l1,L2:List,L3:List) = true"
  eq (l1 @ L2:List) @ L3:List = l1 @ (L2 @ L3) .
-- arbitrary element e:Elt
  op e : -> Elt }
--  check whether "@assoc(e | l1,L2:List,L3:List)" is deducible
--  at "APPENDvo-ASSOC-iStep"
--> [01] sub-goal 1 for the goal [0]
red in APPENDvo-ASSOC-iStep : @assoc(e | l1,L2:List,L3:List) .
--> not returns "true"
```

However, the last red command does not return "true" and we need to do case splitting based on the predicate (e = vo).

5.2.1. A proof score for case splitting

A proof score for doing case splitting with (e = vo) can be constructed as follows.

```
**> decide to do case splitting using the predicate (e = vo)
--> case of ((e = vo) = true) i.e. (e = vo)
mod* APPENDvo-ASSOC-iStep-c0(X :: TRIVvo){
pr(APPENDvo-ASSOC-iStep(X))
eq e = vo .}
--> [010] sub-goal 0 for sub-goal [01]
--  check whether "@assoc(e | l1,L2:List,L3:List)" is deducible
--  at APPENDvo-ASSOC-iStep-c0
red in APPENDvo-ASSOC-iStep-c0 : @assoc(e | l1,L2:List,L3:List) .
--> returns "true"
--> case of ((e = vo) = false)
mod* APPENDvo-ASSOC-iStep-c1(X :: TRIVvo){
pr(APPENDvo-ASSOC-iStep(X))
eq (e = vo) = false .}
--> [011] sub-goal 1 for sub-goal [01]
--  check whether "@assoc(e | l1,L2:List,L3:List)" is deducible
--  at APPENDvo-ASSOC-iStep-c1
red in APPENDvo-ASSOC-iStep-c1 : @assoc(e | l1,L2:List,L3:List) .
--> returns "true"
--> QED
```

A model of the module APPENDvo-ASSOC-iStep-c0 is a model of APPENDvo- ASSOC-iStep that satisfies the equation "e = vo" (i.e."(e = vo) = true"). And, a model of the module APPENDvo-ASSOC-iStep-c1 is a model of APPENDvo -ASSOC-iStep that satisfies the equation "(e = vo) = false". Because APPENDvo-ASSOC-iStep is protected, Bool = {true,false} is also protected. It implies that "(e = vo) = true" and "(e = vo) = false" are the two cases which cover all possibilities. Hence, we get the following implication for justifying the proof score, and the proof is over.

$$\frac{\texttt{APPENDvo-ASSOC-iStep-c0} \models \texttt{@assoc(e | l1,L2:List,L3:List)} \quad \texttt{APPENDvo-ASSOC-iStep-c1} \models \texttt{@assoc(e | l1,L2:List,L3:List)}}{\texttt{APPENDvo-ASSOC-iStep} \models \texttt{@assoc(e | l1,L2:List,L3:List)}}$$

6. Specification calculi

Let SP be an equational specification with a constructor-based order-sorted signature (i.e., $\mathbb{S}ig(SP) = (S, \leq, F, F^c)$), which is constructed by the specification building operations **BS**, **SU**, **PR**, and/or **IN** defined in Section 3.5. Notice that we have denoted the class of all the structured specifications by *SPEC*. The goal of constructing proof scores for a specification $SP \in SPEC$ is proving $SP \models e$ for an equation e that describes a property of interest about SP.

For proving properties that only hold for reachable models, induction is necessary. If the specification is described with conditional equations or conditional expressions (e.g. `if` expressions), case splitting is also necessary. For doing induction and/or case splitting, a goal $SP \models e$ expressed as a satisfaction assertion is decomposed into several sub-goals (that are also expressed as satisfaction assertions) such that their conjunction implies the original goal. These implications (i.e. conjunction $\Rightarrow$ original) used in the decompositions can be formalized as proof rules for constructing proofs. We already saw some applications of this kind of proof rules for induction (Section 2.2.3) or case splitting (Section 5.2.1). Decomposition of a goal expressed as a satisfaction assertion is necessary also for lemma introduction, which is an important step for constructing proofs.

This section first gives a few collections of proof rules that can define specification calculi for proving a semantic assertion $SP \models e$. Based on the developed specification calculi, **the specification calculus for proof score** is given, which is a sufficiently powerful calculus for the constructions of proof scores for proving the semantic assertions.

A specification SP considered in this section is a structured constructor-based order-sorted equational specification unless otherwise stated.

6.1. The basic specification calculus

Definition 17. An entailment system for deducing the logical consequences of the structured specifications is a family of predicates

$$\vdash_{\_} \stackrel{\text{def}}{=} \{SP \vdash_{\_}\}_{SP \in SPEC}$$

on the sets of equations that satisfies the following rules that are called **the specification entailment rules**.

$$[\text{axiom-s}]\ \frac{E_0 \subseteq E}{(\Sigma, E) \vdash E_0} \qquad [\text{union-s}]\ \frac{SP \vdash E_1 \quad SP \vdash E_2}{SP \vdash E_1 \cup E_2}$$

$$[\text{lemma-s}]\ \frac{SP + (\mathbb{S}ig(SP), E_0) \vdash E \quad SP \vdash E_0}{SP \vdash E}$$

$$[\text{trans-s}]\ \frac{SP_0 \subseteq SP \quad SP_0 \vdash E}{SP \vdash E}. \quad \square$$

Notice that $\models_{\_} \stackrel{\text{def}}{=} \{SP \models_{\_}\}_{SP \in SPEC}$ is an example of entailment system for structured specifications. See Definition 4 for $SP \models E$.

A syntactic entity like $SP \vdash E$ is called a specification entailment or a $(SP \vdash E)$-type entailment. A calculus that involves rules composed of the $(SP \vdash E)$-type entailments is called a **specification calculus**.

Definition 18. The basic specification calculus is defined as the least entailment system that satisfies the specification entailment rules (i.e [axiom-s], [union-s], [lemma-s], and [trans-s]) and the following eq rule.

$$[\text{eq}]\ \frac{E \mid\!\!\vdash_{\Sigma} t = t' \quad Flat(SP) = (\Sigma, E)}{SP \vdash t = t'}.$$

Here t, t' are ground terms, and the relation $E \mid\!\!\vdash_{\Sigma} E'$ is defined as the least relation[13] that satisfies the equational rules (i.e. [reflexivity], [symmetry], [transitivity], [congruence], and [instantiation] with $\mid\!\!\vdash_{\Sigma}$ instead of $\vdash_{\Sigma}$, see Definition 11). $\square$

Proposition 19. *The basic specification calculus is sound, i.e. $SP \vdash E$ implies $SP \models E$, for all specifications SP and sets E of $\mathbb{S}ig(SP)$-equations.*

Proof. We show that $\{SP \vdash_{\_}\}_{SP \in SPEC}$ is closed to the rule [eq]. Let $E \subseteq \mathbb{S}en(\Sigma)$, and $t, t' \in T_{\Sigma}$. Assume $E \mid\!\!\vdash_{\Sigma} t = t'$ and $Flat(SP) = (\Sigma, E)$, for a specification SP. Since $\_\mid\!\!\vdash_{\_} \subseteq \_\vdash_{\_} \subseteq \_\models_{\_}$, we have $E \mid\!\!\vdash_{\Sigma} t = t'$ implies $E \models_{\Sigma} t = t'$, and we get $E \models_{\Sigma} t = t'$, and hence $(\Sigma, E) \models t = t'$. Since $Flat(SP) = (\Sigma, E)$, we have $Mod(SP) \subseteq \text{Mod}(\Sigma, E)$, which implies $SP \models t = t'$.

Since $\{SP \vdash_{\_}\}_{SP \in SPEC}$ is the least entailment system closed to [eq], we obtain $SP \vdash E$ implies $SP \models E$ for all $SP \in SPEC$ and $E \subseteq \mathbb{S}en(\mathbb{S}ig(SP))$. $\square$

13 Notice that this relation is not an entailment system.

Proposition 20. *For any sets E and E_0 of Σ-equations, if $E \vdash_\Sigma E_0$ in the basic equational calculus, then $(\Sigma, E) \vdash E_0$ in the basic specification calculus.*

Proof. By induction on the definition of $E \vdash_\Sigma E_0$.

1. For the [lemma-e] case if $E \cup E_0 \vdash_\Sigma E_1$ and $E \vdash_\Sigma E_0$ then by induction hypothesis, $(\Sigma, E \cup E_0) \vdash E_1$ and $(\Sigma, E) \vdash E_0$. Since $(\Sigma, E) + (\Sigma, E_0)$ is a refinement of $(\Sigma, E \cup E_0)$, by [trans-s] $(\Sigma, E) + (\Sigma, E_0) \vdash E_1$, and by [lemma-s], $(\Sigma, E) \vdash E_1$. The remaining cases [axiom-e], [trans1-e] and [trans2-e] are similar.
2. For the [congruence] case, we have $\{(\forall X) t_i = t_i' \mid i \in \overline{1..n}\} \mid\!\vdash (\forall X) f(t_1, \ldots, t_n) = f(t_1', \ldots, t_n')$, and by [eq] we obtain $(\Sigma, \{(\forall X) t_i = t_i' \mid i \in \overline{1..n}\}) \vdash (\forall X) f(t_1, \ldots, t_n) = f(t_1', \ldots, t_n')$. The remaining cases [reflexivity], [symmetry], [transitivity], and [instantiation] are similar. □

6.2. The constructor-based specification calculus

Notice that, in the basic specification calculus, "$SP \vdash e$ implies $SP \models e$" but in general "$SP \models e$ does not imply $SP \vdash e$", because $\text{Mod}(SP)$ contains only reachable models for a constructor-based specification SP and an equation e.

In this section, the "Quasi Completeness Theorem" (Theorem 16) is lifted up to the structured specification level. For that, we are going to lift up each of the equational constructor rules in Definition 14 (i.e. [imp1-e], [imp2-e], [subst-e], [thConst1-e], [thConst2-e], and [abst-e]) to the structured specification level in the Sections 6.2.1–6.2.4. After that, we define the constructor-based specification calculus and give a quasi completeness theorem for that in the Section 6.2.5.

6.2.1. Implication

The following two rules show that conditional equations and unconditional equations are interchangeable.

$$[\text{imp1-s}]\ \frac{SP \vdash t = t' \text{ if } \{t_1 = t_1', \ldots, t_n = t_n'\}}{SP + (\text{Sig}(SP), \{t_1 = t_1', \ldots, t_n = t_n'\}) \vdash t = t'}$$

$$[\text{imp2-s}]\ \frac{SP + (\text{Sig}(SP), \{t_1 = t_1', \ldots, t_n = t_n'\}) \vdash t = t'}{SP \vdash t = t' \text{ if } \{t_1 = t_1', \ldots, t_n = t_n'\}}.$$

Here SP is any specification, and t, t', t_i, t_i' are ground terms. Notice that the conditional equation in the rule is quantifier free (i.e. no variables are declared).

Lemma 21. *The rules [imp1-s] and [imp2-s] are sound.*

Proof. Assume $SP \models t = t' \text{ if } \{t_1 = t_1', \ldots, t_n = t_n'\}$, and we prove $SP + (\text{Sig}(SP), \{t_1 = t_1', \ldots, t_n = t_n'\}) \models t = t'$.

Let $M \in \text{Mod}(SP + (\text{Sig}(SP), \{t_1 = t_1', \ldots, t_n = t_n'\}))$. We have $M \in \text{Mod}(SP)$ and $M \models_{\text{Sig}(SP)} t_i = t_i'$ for all $i \in \overline{1..n}$. Since $SP \models t = t' \text{ if } \{t_1 = t_1', \ldots, t_n = t_n'\}$, we obtain $M \models_{\text{Sig}(SP)} t = t' \text{ if } \{t_1 = t_1', \ldots, t_n = t_n'\}$, which implies $M \models_{\text{Sig}(SP)} t = t'$.

For the converse implication, we assume $SP + (\text{Sig}(SP), \{t_1 = t_1', \ldots, t_n = t_n'\}) \models t = t'$, and we prove $SP \models t = t' \text{ if } \{t_1 = t_1', \ldots, t_n = t_n'\}$. Let $M \in \text{Mod}(SP)$ such that $M \models_{\text{Sig}(SP)} t_i = t_i'$ for all $i \in \overline{1..n}$. We obtain $M \in \text{Mod}(SP + (\text{Sig}(SP), \{t_1 = t_1', \ldots, t_n = t_n'\}))$ and $M \models_{\text{Sig}(SP)} t = t'$. □

6.2.2. Theorem of constants

The following two rules formalize the theorem of constants, which shows that variables and constants are interchangeable.

$$[\text{thConst1-s}]\ \frac{SP \vdash (\forall Y)\varepsilon}{\text{PR}(SP, \Sigma(Y)) \vdash \varepsilon} \qquad [\text{thConst2-s}]\ \frac{\text{PR}(SP, \Sigma(Y)) \vdash \varepsilon}{SP \vdash (\forall Y)\varepsilon}.$$

Here in the universal quantification Y is an S-sorted set of variables, whereas Y in $\text{PR}(SP, Y)$ is considered to be an S-sorted set of fresh constants (i.e. there is no name clash between $\text{Sig}(SP)$ and Y).

Lemma 22. *The rules [thConst1] and [thConst2] are sound.*

Proof. Assume $SP \models (\forall Y)\varepsilon$, and we prove $\text{PR}(SP, \Sigma(Y)) \models \varepsilon$. Let $M' \in \text{Mod}(\text{PR}(SP, \Sigma(Y)))$. Since $M'|_\Sigma \in \text{Mod}(SP)$ and $SP \models (\forall Y)\varepsilon$, we have $M'|_\Sigma \models_\Sigma (\forall Y)\varepsilon$, which implies $M' \models_{\Sigma(Y)} \varepsilon$.

For the converse implication, assume $\text{PR}(SP, \Sigma(Y)) \models \varepsilon$, and we prove $SP \models (\forall Y)\varepsilon$. Let $M \in \text{Mod}(SP)$ and $f : Y \to M$. Since $(M, f) \in \text{PR}(SP, \Sigma(Y))$ and $\text{PR}(SP, \Sigma(Y)) \models \varepsilon$, we have $(M, f) \models_{\Sigma(Y)} \varepsilon$. □

6.2.3. Substitutivity

The following rule shows that new equations can be deduced by substituting terms for variables.

$$[\text{subst-s}]\ \frac{SP \vdash (\forall X)\ \varepsilon}{SP \vdash (\forall Y)\ \theta(\varepsilon)}.$$

Here $(\forall X)\varepsilon$ is a $\mathbf{Sig}(SP)$-equation, and $\theta : X \to T_{\mathbf{Sig}(SP)}(Y)$ is any $\mathbf{Sig}(SP)$-substitution.

Lemma 23. *The rule [*subst-s*] is sound.*

Proof. Let $M \in \text{Mod}(\mathbf{Sig}(SP), \{(\forall X)\varepsilon\})$. By Proposition 15, $(\forall X)\varepsilon \models_{\mathbf{Sig}(SP)} (\forall Y)\theta(\varepsilon)$, which implies $M \models_{\mathbf{Sig}(SP)} (\forall Y)\theta(\varepsilon)$. □

6.2.4. Abstraction

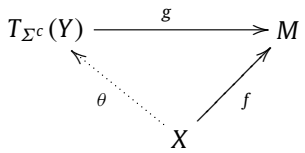
Let SP be a specification, and $(\forall X)\varepsilon$ a $\mathbf{Sig}(SP)$-equation such that the variables X are of constrained sorts. We define the following proof rule for the equations quantified over variables of constrained sorts:

$$[\text{abst-s}]\ \frac{\{\ SP \vdash (\forall Y)\theta(\varepsilon)\ \mid\ \theta : X \to T_{\Sigma^c}(Y),\ Y\text{-finite set of loose variables}\ \}}{SP \vdash (\forall X)\varepsilon}.$$

Here $\Sigma = \mathbf{Sig}(SP)$, $\Sigma^c \subseteq \Sigma$ is the sub-signature of constructors, and $\theta : X \to T_{\Sigma^c}(Y)$ range over all substitutions of constructor terms (i.e. terms formed with constructors and variables of loose sorts) for the variables in X. The above rule says that if ε holds for all possible instantiations of the variables in X with constructor terms then $(\forall X)\ \varepsilon$ is deduced. Note that the premises of this rule can be infinite.

Lemma 24. *The rule [*abst-s*] is sound.*

Proof. Assume $SP \models (\forall Y)\theta(\varepsilon)$ for all substitutions $\theta : X \to T_{\Sigma^c}(Y)$, where Y is a finite set of variables of loose sorts, and we prove $SP \models (\forall X)\varepsilon$. Let $M \in \text{Mod}(SP)$ and $f : X \to M$. Since M is a reachable, there exists a set Y of variables of loose sorts, and a valuation $g : Y \to M$ such that for all $s \in S^c$ the function $g^{\#}_s : (T_{\Sigma^c}(Y))_s \to M_s$ is surjective, where S^c is the subset of constrained sorts, $\Sigma^c \subseteq \Sigma = \mathbf{Sig}(SP)$ is the sub-signature of constructors, and $g^{\#} : T_{\Sigma^c}(Y) \to M$ is the unique extension of g to a Σ^c-morphism.



Since g is surjective on constrained sorts and X consists only constrained sorts, there exists $\theta : X \to T_{\Sigma^c}(Y)$ such that $\theta; g = f$. Because X is finite, there exists $Y_0 \subseteq Y$ finite such that $\theta; g_0 = f$, where $g_0 = g|_{Y_0}$ is the restriction of g to Y_0. Since $SP \models (\forall Y_0)\theta(\varepsilon)$ and $M \in \text{Mod}(SP)$, we have $(M, g_0) \models_{\Sigma(Y_0)} \varepsilon$, which implies $(M, \theta; g_0) \models_{\Sigma(X)} \varepsilon$. That is $(M, f) \models_{\Sigma(X)} \varepsilon$. □

6.2.5. The quasi completeness theorem

Definition 25. The constructor-based specification calculus is the least entailment system that includes the basic specification calculus and satisfies the rules [imp1-s], [imp2-s], [thConst1-s], [thConst2-s], [subst-s], and [abst-s] that are called **the specification constructor rules**. □

Proposition 26 (*Soundness*)**.** *The constructor-based specification calculus is sound.*

Proof. By Proposition 19, $\{SP \models\_\}_{SP \in SPEC}$ is closed to the rule [eq]. By Lemmas 21–24, $\{SP \models\_\}_{SP \in SPEC}$ is closed to the rules [imp1-s], [imp2-s], [subst-s], [thConst1-s], [thConst2-s], and [abst-s]. Because $\{SP \vdash\_\}_{SP \in SPEC}$ is the least entailment system satisfying these properties, for all specifications SP, we have $SP \vdash\_ \subseteq SP \models\_$. □

Proposition 27. *For any sets E and E_0 of Σ-equations, if $E \vdash_\Sigma E_0$ in the constructor-based equational calculus, then $(\Sigma, E) \vdash E_0$ in the constructor-based specification calculus.*

Proof. By induction on the definition of $E \vdash_\Sigma E_0$.

1. The cases [axiom-e], [union-e], [lemma-e], [trans1-e], [trans2-e], [reflexivity], [symmetry], [transitivity], [congruence], and [instantiation] are the same as in the Proposition 20.
2. For the case [abst-e], we assume that $E \vdash_\Sigma (\forall Y)\theta(\varepsilon)$ for all substitutions $\theta : X \to T_{\Sigma^c}(Y)$, where Y is a finite set of variables of loose sorts. We have $E \vdash_\Sigma (\forall X)\varepsilon$, and we need to prove $(\Sigma, E) \vdash (\forall X)\varepsilon$. By the induction hypothesis we have $(\Sigma, E) \vdash (\forall Y)\theta(\varepsilon)$ for all substitutions $\theta : X \to T_{\Sigma^c}(Y)$, where Y is a finite set of variables of sort loose. By the rule [abst-s], for specifications we obtain $(\Sigma, E) \vdash (\forall X)\varepsilon$. The remaining cases, [thConst2-s], [thConst1-s], [subst-s], [imp2-s], and [imp1-s] are similar. □

Completeness of equational calculus constitutes one of the fundamental results in algebraic specification and related research has a long tradition beginning with Birkhoff [24], continuing with the completeness of many-sorted equational deduction [25], and order-sorted equational deduction [18]. Here, we prove a completeness result that lifts the completeness in [15] at the level of structured specifications. A similar result has been reported in [26], but the proof rules (and the entailment system based on those proof rules) do not fit our methodology.

Theorem 28 (*Quasi Completeness Theorem*)**.** *In the constructor-based specification calculus,*

$$SP \models E \text{ implies } SP \vdash E$$

if

1. *SP is sufficiently complete, and*
2. *SP is formed only with the operations* BS, SU, *and* PR.

Proof. Assume that $SP \models E$, and let $(\Sigma, E') = Flat(SP)$. By Fact 5, $\mathrm{Mod}(SP) = \mathrm{Mod}(\Sigma, E')$, which implies $(\Sigma, E') \models E$ and furthermore $E' \models_\Sigma E$. Since SP is sufficiently complete, (Σ, E') is sufficiently complete. By Theorem 16, we obtain $E' \vdash_\Sigma E$, and by Proposition 27, $(\Sigma, E') \vdash E$. By the [trans-s] property of the entailment system for structured specifications, we obtain $SP \vdash E$. □

6.2.6. Structural induction

In order to make proofs it is mandatory to have a finitary procedure to deal with the infinite premises of [abst-s]. The standard one is the so-called **structural induction**. In the following, it is shown that the induction scheme given in [27] can be lifted up to the structured specification level, and hence it is a part of the constructor-based specification calculus.

Assume that we need to prove a property $(\forall X)\varepsilon$ for a specification SP. Let $\mathtt{CON}$ be a sort-preserving mapping $X \to F^c$,[14] where F^c consists of all constructors in $\Sigma = Sig(SP)$ for the sorts of the variables in X. For each $x \in X$, let $Z_{x,\mathtt{CON}} = z^1_{x,\mathtt{CON}} \ldots z^n_{x,\mathtt{CON}}$ be a string of arguments for the constructor $\mathtt{CON}(x) \in F^c$. We define the set of variables $Z_{\mathtt{CON}} = \cup_{x\in X} Z_{x,\mathtt{CON}}$,[15] the substitution $\mathtt{VAR}_{\mathtt{CON}}{}^{\#} : X \to T_\Sigma(Z_{\mathtt{CON}})$ by $\mathtt{VAR}_{\mathtt{CON}}{}^{\#}(x) = \mathtt{CON}(x)(Z_{x,\mathtt{CON}})$ and we let $\mathtt{VAR}_{\mathtt{CON}}$ range over all sort-preserving mappings $X \to T_{\Sigma^c}(Z_{\mathtt{CON}})$ with the following two properties:

1. for all $x \in X$ we have $\mathtt{VAR}(x) \in Z_{x,\mathtt{CON}}$ or $\mathtt{VAR}(x) = \mathtt{CON}(x)(Z_{x,\mathtt{CON}})$, and
2. there exists $x \in X$ such that $\mathtt{VAR}(x) \in Z_{x,\mathtt{CON}}$.

The function $\mathtt{CON}$ will give all the induction cases, while the function $\mathtt{VAR}_{\mathtt{CON}}$ is used to define the induction hypothesis for each case. The induction scheme is defined as follows:

$$[\mathsf{ind}]\ \frac{\{SP_{\mathtt{CON}} \vdash \mathtt{VAR}_{\mathtt{CON}}{}^{\#}(\varepsilon) \mid \mathtt{CON} : X \to F^c\}}{SP \vdash (\forall X)\varepsilon}$$

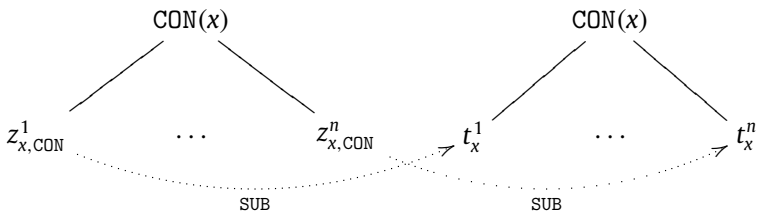
where $SP_{\mathtt{CON}}$ is obtained by adding the induction hypothesis, i.e.

$$SP_{\mathtt{CON}} \stackrel{\mathrm{def}}{=} \mathrm{PR}(SP,\ \Sigma(Z_{\mathtt{CON}})) + (\Sigma(Z_{\mathtt{CON}}), \{\mathtt{VAR}_{\mathtt{CON}}(\varepsilon) \mid \mathtt{VAR}_{\mathtt{CON}} : X \to T_{\Sigma^c}(Z_{\mathtt{CON}})\}).$$

Proposition 29. *The constructor-based specification calculus is closed to (or satisfies) [*ind*].*

Proof. Assume that $SP_{\mathtt{CON}} \vdash \mathtt{VAR}_{\mathtt{CON}}{}^{\#}(\varepsilon)$ for all $\mathtt{CON} : X \to F^c$. If we prove $SP \vdash (\forall Y)\theta(\varepsilon)$ for all substitutions $\theta : X \to T_{\Sigma^c}(Y)$, where Y is a finite set of variables of loose sorts, then by [abst-s] we obtain $SP \vdash (\forall X)\varepsilon$. In the following we focus on proving this assertion.

We proceed by induction on the sum of the depth of the terms in $\{\theta(x) \mid x \in X\}$, which exists as a consequence of X being finite. Let $\mathtt{CON} : X \to F^c$ be the sort-preserving mapping such that for all $x \in X$, $\mathtt{CON}(x)$ is the topmost constructor of $\theta(x)$. Let $T_x = t^1_x \ldots t^n_x$ be the string of the immediate sub-terms of $\theta(x)$, and $Y = \cup_{x\in X} Y_x$, where Y_x are all variables of the terms in T_x. We define the substitution $\mathtt{SUB} : Z_{\mathtt{CON}} \to T_{\Sigma^c}(Y)$ by $\mathtt{SUB}(z^i_{x,\mathtt{CON}}) = t^i_x$.



Note that

$$\mathrm{PR}(SP,\ \Sigma(Y)) + (\Sigma(Y), \{\mathtt{VAR}_{\mathtt{CON}}; \mathtt{SUB}(\varepsilon) \mid \mathtt{VAR}_{\mathtt{CON}} : F^c \to Z_{\mathtt{CON}}\})$$

14 $\mathtt{CON}(x)$ have the same sort as x.

15 Here, $Z_{x,\mathtt{CON}}$ is regarded as a set of variables.

is a refinement of

$$\mathrm{PR}(SP,\ \Sigma(Z_{\mathrm{CON}})) + (\Sigma(Z_{\mathrm{CON}}), \{\mathrm{VAR}_{\mathrm{CON}}(\varepsilon) \mid \mathrm{VAR}_{\mathrm{CON}} : F^c \to Z_{\mathrm{CON}}\}).$$

By the premises of [ind] the following holds

$$\mathrm{PR}(SP,\ \Sigma(Z_{\mathrm{CON}})) + (\Sigma(Z_{\mathrm{CON}}), \{\mathrm{VAR}_{\mathrm{CON}}(\varepsilon) \mid \mathrm{VAR}_{\mathrm{CON}} : F^c \to Z_{\mathrm{CON}}\}) \vdash {\mathrm{VAR}_{\mathrm{CON}}}^{\#}(\varepsilon).$$

By [trans-s] property of the entailment systems we obtain

$$\mathrm{PR}(SP,\ \Sigma(Y)) + (\Sigma(Y), \{(\mathrm{VAR}_{\mathrm{CON}}; \mathrm{SUB})(\varepsilon) \mid \mathrm{VAR}_{\mathrm{CON}} : F^c \to Z_{\mathrm{CON}}\}) \vdash (\mathrm{VAR}^{\#}_{\mathrm{CON}}; \mathrm{SUB})(x).$$

For all $x \in X$ we have $\theta(x) = (\mathrm{VAR}^{\#}_{\mathrm{CON}}; \mathrm{SUB})(x)$, which implies $\theta(\varepsilon) = (\mathrm{VAR}^{\#}_{\mathrm{CON}}; \mathrm{SUB})(\varepsilon)$. We get

$$\mathrm{PR}(SP,\ \Sigma(Y)) + (\Sigma(Y), \{(\mathrm{VAR}_{\mathrm{CON}}; \mathrm{SUB})(\varepsilon) \mid \mathrm{VAR}_{\mathrm{CON}} : F^c \to Z_{\mathrm{CON}}\}) \vdash \theta(x).$$

The sum of the depth of the terms in $\{(\mathrm{VAR}_{\mathrm{CON}}; \mathrm{SUB})(x) \mid x \in X\}$ is strictly less than the sum of the depth of the terms in $\{\theta(x) \mid x \in X\}$; hence we can apply the induction hypothesis and we get $SP \vdash (\forall Y)(\mathrm{VAR}_{\mathrm{CON}}; \mathrm{SUB})(\varepsilon)$ for all $\mathrm{VAR}_{\mathrm{CON}} : X \to T_{\Sigma^c}(Z_{\mathrm{CON}})$. By [thConst1-s], $\mathrm{PR}(SP,\ \Sigma(Y)) \vdash (\mathrm{VAR}_{\mathrm{CON}}; \mathrm{SUB})(\varepsilon)$ for all $\mathrm{VAR}_{\mathrm{CON}} : X \to T_{\Sigma^c}(Z_{\mathrm{CON}})$. By [union-s], $\mathrm{PR}(SP,\ \Sigma(Y)) \vdash \{(\mathrm{VAR}_{\mathrm{CON}}; \mathrm{SUB})(\varepsilon) \mid \mathrm{VAR}_{\mathrm{CON}} : X \to T_{\Sigma^c}(Z_{\mathrm{CON}})\}$. By [lemma-s], $\mathrm{PR}(SP,\ \Sigma(Y)) \vdash \theta(\varepsilon)$. By [thConst2-s], $SP \vdash (\forall Y)\theta(\varepsilon)$. □

Example 30. The proof in Section 2.2.3 corresponds to the proof of

$$\mathtt{APPEND\text{-}ASSOC} \vdash (\forall L_1, L_2, L_3 : \mathtt{List})\mathtt{@assoc}(L_1, L_2, L_3)$$

by structural induction over the first variable L_1. Take $SP = \mathtt{APPEND\text{-}ASSOC}$ and $\varepsilon = (\forall L_2, L_3 : \mathtt{List})\mathtt{@assoc}(L_1, L_2, L_3)$. There are two cases: $\mathtt{CON}(L1) = \mathtt{nil}$ and $\mathtt{CON}(L1) = \mathtt{\_|\_}$. For the first case, $\mathtt{APPEND\text{-}ASSOC}_{\mathrm{CON}} = \mathtt{APPEND\text{-}ASSOC}$, and we prove $\mathtt{APPEND\text{-}ASSOC} \vdash (\forall L_2, L_3 : \mathtt{List})\mathtt{@assoc}(\mathtt{nil}, L_2, L_3)$. For the second case, let $z^1 z^2$ be the arguments of $\mathtt{\_|\_}$, where z^1 has the sort $\mathtt{Elt}$ and z^2 has the sort $\mathtt{List}$. There is one function $\mathtt{VAR}$ defined by $\mathtt{VAR}(\mathtt{nil}) = z^2$ and $\mathtt{VAR}(\mathtt{\_|\_}) = z^2$. Notice the following.

$$\begin{array}{ll} (\mathtt{CON}; \mathtt{VAR})(L_1) = z^2 & \\ & \mathrm{PR}(\mathtt{APPEND\text{-}ASSOC}, \mathit{Sig}(\mathtt{APPEND\text{-}ASSOC})(z^1, z^2)) + \\ \mathtt{APPEND\text{-}ASSOC}_{\mathrm{CON}} = & (\mathit{Sig}(\mathtt{APPEND\text{-}ASSOC})(z^1, z^2), \\ & \quad (\forall L_2, L_3 : \mathtt{List})\mathtt{@assoc}(z^2, L_2, L_3)) \end{array}$$

We prove $\mathtt{APPEND\text{-}ASSOC}_{\mathrm{CON}} \vdash (\forall L_2, L_3 : \mathtt{List})\mathtt{@assoc}(z^1 \mid z^2, L_2, L_3)$ and the proof is over. In the above proof the following proof rule that is an instance of the rule [ind] is used.

$$\frac{\begin{array}{ll} \mathtt{APPEND\text{-}ASSOC} \vdash & \mathtt{APPEND\text{-}ASSOC}_{\mathrm{CON}} \vdash \\ (\forall L_2, L_3 : \mathtt{List})\mathtt{@assoc}(\mathtt{nil}, L_2, L_3) & (\forall L_2, L_3 : \mathtt{List})\mathtt{@assoc}(z^1 \mid z^2, L_2, L_3) \end{array}}{\mathtt{APPEND\text{-}ASSOC} \vdash (\forall L_1, L_2, L_3 : \mathtt{List})\mathtt{@assoc}(L_1, L_2, L_3)}$$

The above is the proof rule used in Section 2.2.3 with the correspondences $\mathtt{APPEND\text{-}ASSOC}_{\mathrm{CON}} \leftrightarrow \mathtt{APPEND\text{-}ASSOC\text{-}iStep}$, $z^1 \leftrightarrow \mathtt{e}$, and $z^2 \leftrightarrow \mathtt{l1}$. □

6.3. The enhanced constructor-based specification calculus

We have proved the quasi completeness result for the sufficiently complete specifications formed only with BS, SU and PR (see Theorem 28), but, in general, the constructor-based specification calculus is not complete.

In this section the constructor-based specification calculus is enhanced by defining another two important and useful proof rules.

6.3.1. Case splitting

We define a proof rule that divides a goal into a sufficient number of separate cases. Consider a specification SP, and a term $u \in T_{\mathit{Sig}(SP)}$ of a constrained sort. If u cannot be "reduced" to a term formed with constructors and elements of loose sorts (variables or operations) then we need to make a case analysis on the possible values of u. Notice that this rule can be applied to the specifications that may or may not be sufficiently complete.

$$[\mathsf{split}]\ \frac{\{\ \mathrm{PR}(SP,\ \Sigma(Y)) + (\Sigma(Y), \{u = t\}) \vdash E \quad \mid\ t \in T_{\Sigma^c}(Y), Y\text{-finite set of loose variables}\ \}}{SP \vdash E}$$

where $\Sigma = \mathit{Sig}(SP)$, $\Sigma^c \subseteq \Sigma$ is the sub-signature of constructors, and t range over all constructor terms (i.e. terms formed with constructors and variables of loose sorts).

Lemma 31. *The rule [*split*] is sound.*

Proof. Assume that $\mathrm{PR}(SP, \Sigma(Y)) + (\Sigma(Y), \{u = t\}) \models E$ for all constructor terms $t \in T_{\Sigma^c}(Y)$, and we prove $SP \models E$, where $\Sigma = Sig(SP)$ and $\Sigma^c \subseteq \Sigma$ is the sub-signature of constructors.

Let $M \in \mathrm{MOD}(SP)$. Since M is reachable, there exists a set Y of variables of loose sorts, a valuation $f : Y \to M$, and a term $t \in T_{\Sigma^c}(Y)$ such that $f^{\#}(t) = M_u$, where $f^{\#} : T_{\Sigma^c}(Y) \to M$ is the unique extension of f to a Σ^c-morphism. Note that $(M, f) \models u = t$, which implies $(M, f) \in \mathrm{MOD}(\mathrm{PR}(SP, \Sigma(Y)) + (\Sigma(Y), \{u = t\}))$. Since $\mathrm{PR}(SP, \Sigma(Y)) + (\Sigma(Y), \{u = t\}) \models E$, we obtain $(M, f) \models E$, and furthermore $M \models E$. □

The set of terms t above may be infinite, and therefore the premises of [split] may be infinite too. In many examples, the case analysis is conducted on the possible values of a term of the sort `Bool`. When u is a ground term of the sort `Bool`, the possible values of u are just `true` and `false`.

$$[\mathrm{splitBool}]\ \frac{\{\ (\mathrm{PR}(SP, \Sigma) + (\Sigma, \{u = b\})) \vdash E \quad \mid\ b \in \{\mathtt{true}, \mathtt{false}\}\ \}}{SP \vdash E}.$$

Example 32. The proof rule in Section 5.2.1 was an instance of the [splitBool] rule with the following correspondences.

$SP \leftrightarrow$ `APPENDvo-ASSOC-iStep`
$\Sigma \leftrightarrow Sig($`APPENDvo-ASSOC-iStep`$)$
$(\mathrm{PR}(SP, \Sigma) + (\Sigma, \{u =$ `true`$\})) \leftrightarrow$ `APPENDvo-ASSOC-iStep-c0`
$(\mathrm{PR}(SP, \Sigma) + (\Sigma, \{u =$ `false`$\})) \leftrightarrow$ `APPENDvo-ASSOC-iStep-c1`
$u \leftrightarrow$ `(e = vo)`
$E \leftrightarrow$ `{@assoc(e | l1,L2:List,L3:List)}`. □

In some cases we need to iterate the process of splitting the goals with [split] several times. Therefore, it is difficult to formulate a completeness result that does not depend on sufficient completeness.

6.3.2. Initiality

Let specification SP include the module `BOOL` with the sort `Bool`, which is composed of the two distinctive elements `true` and `false`. Notice that `true` and `false` are defined to be distinctive by the initiality declaration using the operator **IN**. (see the Sections 2.1.1 and 5.1). Then we have the following proof rules which state that if contradiction (`true = false`) happens then any equation is deducible.

$$[\mathrm{init}]\ \frac{SP \vdash \mathtt{true} = \mathtt{false}}{SP \vdash E}.$$

Lemma 33. *The rule [*init*] is sound.*

Proof. If $SP \models$ `true = false` then $\mathrm{MOD}(SP) = \emptyset$. Hence $SP \models E$, for all $E \subseteq \mathbb{S}en(Sig(SP))$. □

6.3.3. The soundness of the enhanced calculus

By formalizing the last three rules [ind], [split], and [init], we have obtained a sufficiently powerful set of proof rules for proving a semantic assertion $SP \models e$ for a structured constructor-based order-sorted equational specification SP and an equation e.

Definition 34. The enhanced constructor-based specification calculus is defined as the least entailment system that includes the constructor-based specification calculus and satisfies the rules [ind], [split], and [init]. □

The following soundness result is a direct consequence of Proposition 29, Lemmas 31 and 33.

Theorem 35. *The enhanced constructor-based specification calculus is sound.* □

6.4. The specification calculus for proof scores

In this section, a specification calculus for proof scores is defined based on the enhanced specification calculus. For doing that, (1) the function of CafeOBJ's reduction is defined as a proof rule [red] that partially simulates the rule [eq], and (2) a specification calculus is defined as the least entailment system that satisfies a subset of the developed proof rules plus the rule [red].

For a ground term t, a reduction command "`red in` SP `:` t `.`" of CafeOBJ applies all the equations[16] of SP as rewriting rules from left to right as much as possible and gets a normal form of t. It amounts to applying the proof rules [reflexivity], [transitivity], [congruence], and [instantiation] as much as possible. Notice that the rule [symmetry] is dropped. This implies that the CafeOBJ reduction command is a correct but partial implementation of the equational rules.

[16] More precisely, all the equations declared with the `eq` or `cq` keywords "modulo" the equational axioms of associativity and/or commutativity and/or identity declared as equational attributes of some of the operators in $Sig(SP)$.

Let $flat(SP) = (\Sigma, E)$ and $E \mid\!\vdash_\Sigma t \overset{\text{red}}{\rightarrow\!\leftarrow} t'$, for ground terms $t, t' \in T_\Sigma$, denote the fact that the CafeOBJ reduction command "`red in` $SP : t = t'$." returns `true`. Recall that the relation defined by $E \mid\!\vdash_\Sigma t = t'$ in the rule [eq] is the least relation that satisfies the equational rules (i.e. [reflexivity], [symmetry], [transitivity], [congruence], and [instantiation], see Definition 18). Notice also that the CafeOBJ reduction command "`red in` $SP : t = t'$." simulates $E \mid\!\vdash_\Sigma t = t'$ correctly except for the rule [symmetry]. Hence we have ($E \mid\!\vdash_\Sigma t \overset{\text{red}}{\rightarrow\!\leftarrow} t'$ implies $E \mid\!\vdash_\Sigma t = t'$), and the following proof rule [red] is proved to be sound.

$$[\text{red}]\ \frac{E \mid\!\vdash_\Sigma t \overset{\text{red}}{\rightarrow\!\leftarrow} t' \quad Flat(SP) = (\Sigma, E)}{SP \vdash t = t'}.$$

Notice that if the term rewriting system (TRS) that corresponds to SP is confluent and terminating, as well known, ($E \mid\!\vdash_\Sigma t \overset{\text{red}}{\rightarrow\!\leftarrow} t'$ iff $E \mid\!\vdash_\Sigma t = t'$) holds. Hence the following fact.

Fact 36. *If the TRS that corresponds to a specification SP is confluent and terminating, the proof rules [*eq*] and [*red*] are interchangeable.*

Definition 37. The specification calculus for proof scores is defined as the least entailment system that satisfies the specification entailment rules (i.e. [axiom-s], [union-s], [lemma-s], and [trans-s]), and satisfies the rules [red], [imp1-s], [imp2-s], [thConst1-s], [thConst2-s], [subst-s], [ind], [split] and [init].

Notice the following.

- The rules [reflexivity], [symmetry], [transitivity], [congruence], [instantiation], and [eq] are dropped, because [eq] is replaced by [red] and it does not invoke the other equational rules any more.
- The rule [abst-s] is dropped, because this rule is an infinite rule and supposed to be simulated by the rule [ind].

As a corollary of Theorem 35 the following holds.

Corollary 38 (*Soundness*)**.** *The specification calculus for proof scores is sound.* □

6.5. Proof trees and proof scores

For a constructive definition of the deductions by the specification calculus for proof scores, a constructive definition of **proof trees** is introduced.

Definition 39. Nodes, trees, p-trees (proof trees), roots, and sub-trees are defined recursively as follows.

(T1) A syntactic item like "$E \mid\!\vdash_\Sigma t \overset{\text{red}}{\rightarrow\!\leftarrow} t'$", "$E_0 \subseteq E$", "$SP_0 \subseteq SP$", "$Flat(SP) = (\Sigma, E)$", or "$SP \vdash t = t'$" is a **node** which is called a rd-node, an es-node, an ss-node, a fl-node, or an sp-node, respectively. A node n is a **tree**, and n is called the **root** of the tree.

(T2) Let n be a node, $tr_1, \ldots, tr_i$ be trees, and $n_1, \ldots, n_i$ be the roots of the trees $tr_1, \ldots, tr_i$, then $(\{tr_1, \ldots, tr_i\}, n)$ is a **tree** if $\dfrac{n_1, \ldots, n_i}{n}$ is an instance of one of the proof rules of the specification calculus for proof scores. n is called the **root** of the tree, and $tr_1, \ldots, tr_i$ are called the **sub-trees** of the tree or the node n. The sub-tree relation is transitive, and hence if tr_a is a sub-tree of tr_b and tr_b is a sub-tree of tr_c then tr_a is a sub-tree of tr_c.

(T3) A sub-tree of a tree is a leaf if it is a node. A tree is a **p-tree** if none of its leaves is an sp-node. □

Notice that, by the above definition of trees, a tree can be constructed effectively by applying the proof rules recursively.[17]

For constructing a p-tree, a sp-node leaf (i.e. a sp-node that is a leaf) should be eliminated by applying an appropriate proof rule until every leaf becomes a rd-node, an es-node, an ss-node, or a fl-node.

Given a predicate p about the specification SP, if we can construct a p-tree whose root is the specification entailment $SP \vdash (p = \texttt{true})$, then the entailment is deduced (or proved) by the specification calculus for proof scores.

Given a p-tree tr, let $\{n_1, \ldots, n_p\}$ denote the set of rd-node leaves of tr, and let "`red in` $SP_j : t_j = t'_j$." denote the reduction command for the rd-node leaf n_j for $j \in \overline{1..p}$. The **proof score** of tr is the set of reduction commands

$$\{\ \texttt{red in}\ SP_j : t_j = t'_j\ .\ \mid j \in \overline{1..p}\ \}$$

that correspond to the set of rd-node leaves of tr. More exactly, a proof score is a collection of the CafeOBJ codes that includes all of the following things of a p-tree.

(1) Each CafeOBJ module for each specification SP that appears at some node of the p-tree.
(2) Each reduction command that corresponds to some rd-node leaf of the p-tree.

The **proof score method** is an interactive verification method for verifying $SP \vdash (p = \texttt{true})$ by constructing a proof score.

17 This construction may be incomplete because it is difficult to enumerate all the instances of rules [split].

7. Conclusions

7.1. Related works and characteristics of the proof score method

Maude [28] is a sister language of CafeOBJ and both languages share many important features. Maude's basic logic is rewriting logic [29] and verification based on Maude mainly focuses on sophisticated model checking with a powerful associative and/or commutative rewriting engine. Maude's sub-language of functional modules is a superset of the order-sorted equational specifications discussed in this paper. For functional modules, the Maude Induction Theorem Prover (ITP) [30] mechanizes inference rules similar to those in our specification calculus [31,32] and provides an attractive method to perform inductive proofs for order-sorted and membership equational specifications.

The most notable theorem proving (or interactive verification) methods and/or systems supporting induction are either first-order or higher-order. First-order such systems include Otter [33], ACL2 [34], and SNARK [35]. Higher-order such systems include Coq [36], Isabelle/HOL [37], and PVS [38]. These lists are no way comprehensive because there are quite many theorem proving methods/systems developed until now, and it is out of the scope of this paper to classify them and discuss their important features. We can argue, however, that there have been quite a few high quality research achievements around these methods/systems, and they are considered to be the most important parts of computer science and software technology.

The proof score method formalized as the specification calculus on the CafeOBJ language system, which is described in this paper, has the following characteristics compared to the already developed theorem proving methods/systems.

- The CafeOBJ language was originally designed as a general purpose formal specification language for a wide variety of systems. And one of the main features of the language is the specification method based on algebraic abstract types that has a high potential to describe specifications in an appropriate abstraction level. The powerful module system (e.g. parameterized modules) is a strong support for this. This characteristic is unique among the languages accompanying to other theorem proving systems that were mainly designed for describing/proving mathematical theorems.
- CafeOBJ is an executable formal specification language, and the specifications in CafeOBJ can be executed by reduction/rewriting for checking/testing properties of interests. A proof score is an important example of the executable CafeOBJ code. This is a unique and important feature of CafeOBJ and proof scores.
- Automated parts of verification are done solely by reduction/rewriting of the CafeOBJ language system which is correct with respect to equational deduction. And the remaining interactive parts are formally modeled as the specification calculus. This two layered structure can provide simple, transparent, but powerful architecture for interactive verification.
- Semantics of verifications are defined based on models that satisfy specifications. The specification calculus is based on this semantics and formalize the verification procedures at the level of goals expressed as satisfaction assertions $SP \models e$. These are unique to the proof score method described in this paper compared to any other verification methods.[18]

7.2. Future issues

There are many researches on the theory and method to check and guarantee that a TRS is terminating, confluent, and/or sufficiently complete[19]. It is an interesting and important research topic to enhance these studies and develop the theory and method to check and guarantee that every specification appearing in a specification calculus is terminating, confluent, and/or sufficiently complete. Here a specification is said to be terminating, confluent, and/or sufficiently complete if the corresponding TRS has the properties. If such theory and method are developed, for example, the rule [red] can be guaranteed to replace the rule [eq] completely (see Fact 36), and the power of the specification calculus for proof scores could increase significantly.

Once the specification calculus for proof score is formalized as in Section 6, applications of these rules can also be described as rewritings and can be coded in CafeOBJ/Maude. That is, constructions of p-trees and proof scores themselves can be specified and analyzed, and/or verified in CafeOBJ/Maude. It can lead to semi-automatic construction of p-trees and proof scores, and is a challenging research topic in the future. As a matter of fact, we have already implemented a prototype proof assistance in Maude meta-programming based on the specification calculus for proof score. The proof assistance can prove, for an example, "the motivating example" of this paper automatically if a user gives a top level proof strategy.

Acknowledgments

The authors appreciate anonymous reviewers' efforts for preparing many valuable comments timely, which have contributed a lot for improving the quality of the paper.

18 The only one exception is that [31] adopted an entailment similar to the specification entailment in their inference system. However, they did not adopt proof scores nor constructor-based specifications.

19 Any term has a normal form which is composed of constructors and variables of loose sorts.

José Meseguer has read an early version of the paper carefully and provided many detailed comments. Răzvan Diaconescu has also commented on an early version of the paper. These have helped to improve the paper significantly.

The first author (KF) appreciates the occasion of preparing this paper for the Festschrift in honor of Yoshihito Toyama. KF has a good memory of working with Toyama-san for writing a Japanese tutorial on TRS in an early age (1983), and is hoping that this paper contributes to "new directions in rewriting".

References

[1] K. Futatsugi, J.A. Goguen, J.-P. Jouannaud, J. Meseguer, Principles of OBJ2, in: Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages, January 1985, POPL85, ACM, 1985, pp. 52–66.
[2] J.A. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, J.-P. Jouannaud, Introducing OBJ, in: J. Goguen, G. Malcolm (Eds.), Software Engineering with OBJ: Algebraic Specification in Action, Kluwer Academic Publishers, 2000.
[3] J. Goguen, Theorem proving and algebra, [Unpublished Book], now being planned to be up on the web for the free use.
[4] R. Diaconescu, K. Futatsugi, CafeOBJ Report: The Language, Proof Techniques, and Methodologies for Object-oriented Algebraic Specification, in: AMAST Series in Computing, World Scientific, 1998.
[5] R. Diaconescu, K. Futatsugi, Logical foundations of CafeOBJ, Theor. Comput. Sci. 285 (2002) 289–318.
[6] CafeOBJ, Web page, http://www.ldl.jaist.ac.jp/cafeobj/, 2012.
[7] K. Futatsugi, A. Nakagawa, An overview of CAFE specification environment - an algebraic approach for creating, verifying, and maintaining formal specifications over networks, in: Proc. of 1st International Conference on Formal Engineering Methods, ICFEM, IEEE, 1997, pp. 170–182.
[8] K. Futatsugi, Formal methods in CafeOBJ, in: Z. Hu, M. Rodríguez-Artalejo (Eds.), FLOPS, in: Lecture Notes in Computer Science, vol. 2441, Springer, 2002, pp. 1–20.
[9] K. Ogata, K. Futatsugi, Proof scores in the OTS/CafeOBJ method, in: Proceedings of the 6th IFIP WG6.1 International Conference on Formal Methods for Open Object-Based Distributed Systems, in: Lecture Notes in Computer Science, vol. 2884, Springer, 2003, pp. 170–184.
[10] K. Futatsugi, J.A. Goguen, K. Ogata, Verifying design with proof scores, in: B. Meyer, J. Woodcock (Eds.), VSTTE, in: Lecture Notes in Computer Science, vol. 4171, Springer, 2005, pp. 277–290.
[11] K. Ogata, K. Futatsugi, Some tips on writing proof scores in the OTS/CafeOBJ method, in: K. Futatsugi, J.-P. Jouannaud, J. Meseguer (Eds.), Essays Dedicated to Joseph A. Goguen, in: Lecture Notes in Computer Science, vol. 4060, Springer, 2006, pp. 596–615.
[12] K. Futatsugi, Verifying specifications with proof scores in CafeOBJ, in: Proc. of 21st IEEE/ACM International Conference on Automated Software Engineering, in: ASE 2006, IEEE Computer Society, 2006, pp. 3–10.
[13] K. Futatsugi, Fostering proof scores in CafeOBJ, in: J.S. Dong, H. Zhu (Eds.), ICFEM, in: Lecture Notes in Computer Science, vol. 6447, Springer, 2010, pp. 1–20.
[14] K. Futatsugi, A. Nakagawa, T. Tamai (Eds.), CAFE: An Industrial-Strength Algebraic Formal Method, Elsevier Science B.V, Amsterdam, The Netherlands, 2000.
[15] D. Găină, K. Futatsugi, K. Ogata, Constructor-based institutions, in: A. Kurz, M. Lenisa, A. Tarlecki (Eds.), CALCO, in: Lecture Notes in Computer Science, vol. 5728, Springer, 2009, pp. 398–412.
[16] J.A. Goguen, R.M. Burstall, Institutions: Abstract model theory for specification and programming, J. ACM 39 (1992) 95–146.
[17] J. Meseguer, Membership algebra as a logical framework for equational specification, in: F. Parisi-Presicce (Ed.), WADT, in: Lecture Notes in Computer Science, vol. 1376, Springer, 1997, pp. 18–61.
[18] J. A. Goguen, J. Meseguer, Order-sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions and partial operations, Theor. Comput. Sci. 105 (1992) 217–273.
[19] D. Sannella, A. Tarlecki, Specifications in an arbitrary institution, Inf. Comput. 76 (1988) 165–210.
[20] R. Diaconescu, Institution-independent Model Theory, first ed., Birkhäuser, Basel, 2008.
[21] D. Găină, K. Futatsugi, Initial semantics in logics with constructors, Journal of Logic and Computation (2012) (in press).
[22] J. Meseguer, J.A. Goguen, Order-sorted algebra solves the constructor-selector, multiple representation, and coercion problems, Inf. Comput. 103 (1993) 114–158.
[23] J. Meseguer, General logics, in: H.-D. Ebbinghaus, et al. (Eds.), Proceedings, Logic Colloquium, 1987, North-Holland, 1989, pp. 275–329.
[24] G. Birkhoff, On the structure of abstract algebras, Proceedings of the Cambridge Philosophical Society 31 (1935) 433–454.
[25] J. Goguen, J. Meseguer, Completeness of many-sorted equational logic, Houston J. Math. 11 (1985) 307–334.
[26] T. Borzyszkowski, Logical systems for structured specifications, Theor. Comput. Sci. 286 (2002) 197–245.
[27] R. Diaconescu, Structural induction in institutions, Inf. Comput. 209 (2011) 1197–1222.
[28] Maude, Web page, http://maude.cs.uiuc.edu/, 2012.
[29] J. Meseguer, Twenty years of rewriting logic, in: P.C. Ölveczky (Ed.), WRLA, in: Lecture Notes in Computer Science, vol. 6381, Springer, 2010, pp. 15–17. an extended version of 120+ pages is submitted for publication.
[30] M. Clavel, M. Palomino, A. Riesco, Introducing the ITP tool: a tutorial, J. UCS 12 (2006) 1618–1650.
[31] M. Clavel, F. Durán, S. Eker, J. Meseguer, Building equational proving tools by reflection in rewriting logic, in: CAFE: An Industrial-Strength Algebraic Formal Method, Elsevier, 2000.
[32] J. Hendrix, D. Kapur, J. Meseguer, Coverset induction with partiality and subsorts: A powerlist case study, in: M. Kaufmann, L.C. Paulson (Eds.), ITP, in: Lecture Notes in Computer Science, vol. 6172, Springer, 2010, pp. 275–290.
[33] Otter, Web page, http://www.cs.unm.edu/~mccune/otter/, 2012.
[34] ACL2, Web page, http://www.cs.utexas.edu/users/moore/acl2/, 2012.
[35] SNARK, Web page, http://www.ai.sri.com/~stickel/snark.html, 2012.
[36] Coq, Web page, http://coq.inria.fr/, 2012.
[37] T. Nipkow, L.C. Paulson, M. Wenzel, Isabelle/HOL: A Proof Assistant for Higher-Order Logic, in: LNCS, vol. 2283, Springer, 2002.
[38] PVS, Web page, http://pvs.csl.sri.com/, 2012.