Foundations of Verification with Proof Scores in CafeOBJ

FUTATSUGI,K. GĂINĂ,D. OGATA,K.

Research Ceneter for Software Verification &
Graduate School of Information Science
JAIST

2 April 2012

- Theoretical principles of proof scores
- Explaining by simple but instructive examples
- Definitions of models and satisfaction relation
- Formalization in the specification calculus
  (a set of proof rules for proof scores)

```
--> 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]}
```

```
--> parametrized 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) .
}
```

$$\begin{array}{lcl}
\mathtt{List} & = & \mathtt{Nil} \cup \mathtt{NnList} \\
\mathtt{Nil} & = & \{\, \mathtt{nil} \,\} \\
\mathtt{NnList} & = & \{\, e \,|\, l \mid e \in \mathtt{Elt},\ l \in \mathtt{List} \}
\end{array}$$

$$\begin{array}{lcl}
\mathtt{List} & = & \{\, \mathtt{nil},\ e_{00} \,|\, \mathtt{nil},\ e_{10} \,|\, e_{11} \,|\, \mathtt{nil},\ \ldots, \\
 & & \quad e_{n0} \,|\, e_{n1} \,|\, \ldots e_{nn} \,|\, \mathtt{nil},\ \ldots \\
 & & \quad \mid e_{ij} \in \mathtt{Elt},\ i, j \in \{0, 1, 2, \ldots\} \,\}
\end{array}$$

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

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

It is assumed that for any sort `*St*` the equality is declared as follows.

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

It guarantees the logical equivalence of CafeOBJ language level (i.e. meta level) equality and sort level (i.e. object level) equality.

```
--> 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)) .
}
```

Verification of associativity of append operation with respect to the specification `APPEND-ASSOC` is formalized as "**verifying that any model of `APPEND-ASSOC` satisfies the following equation**".

$$((\forall L1, L2, L3 : \texttt{List})\texttt{@assoc(}L1, L2, L3\texttt{)} = \texttt{true})$$

This is written as the following **Semantic Assertion**.

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

This can also be written as follows.

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

```
mod* @ASSOC(X :: TRIV){pr(APPEND-ASSOC(X))
-- arbitrary lists l1 l2 l3
ops l1 l2 l3 : -> List }
-- check whether "@assoc(l1,l2,l3)" is deducible
-- at "@ASSOC"
--> [0] the goal
red in @ASSOC : @assoc(l1,l2,l3) .
--> returns "(((l1 @ l2) @ l3) = (l1 @ (l2 @ l3)))"
```

A model M of the module `LIST` interprets the sort `Elt` as a set $M_{\mathtt{Elt}}$, the sort `Nil` as a set $M_{\mathtt{Nil}}$, the sort `NnList` as a set $M_{\mathtt{NnList}}$, the sort `List` as a set $M_{\mathtt{List}}$, the operator `nil` as an operator $M_{\mathtt{nil}}$ `: ->` $M_{\mathtt{Nil}}$, and the operator `_|_` as an operator $\_M_{|}\_$ `:` $M_{\mathtt{Elt}}$ $M_{\mathtt{List}}$ `->` $M_{\mathtt{NnList}}$. A model M of `LIST` is defined to be **reachable** if $M_{\mathtt{List}}$ is represented as follows.

$$\begin{aligned} M_{\mathtt{List}} \;=\; \{\, & M_{\mathtt{nil}},\; e_{00} M_{|} M_{\mathtt{nil}},\; e_{10} M_{|} e_{11} M_{|} M_{\mathtt{nil}},\; \ldots, \\ & e_{n0} M_{|} e_{n1} M_{|} \ldots e_{nn} M_{|} M_{\mathtt{nil}},\; \ldots \\ & \mid e_{ij} \in M_{\mathtt{Elt}}, i, j \in \{0, 1, 2, \ldots\} \,\} \end{aligned}$$

That is, any element of $M_{\mathtt{List}}$ can be constructed with $M_{\mathtt{Elt}}$, $M_{\mathtt{nil}}$, and $\_M_{|}\_$.

```
**> decide to use structural induction w.r.t.
**> the first argument l1 of "@assoc(l1,l2,l3)"
--> Induction base
mod* @ASSOC-iBase(X :: TRIV){pr(@ASSOC(X))}
--  check whether "@assoc(nil,l2,l3)" is deducible
--  at "@ASSOC-iBase"
--> [00] sub-goal 0 for the goal [0]
red in @ASSOC-iBase : @assoc(nil,l2:List,l3:List) .
--> returns "true"
```

```
--> Induction step
mod* @ASSOC-iStep(X :: TRIV){pr(@ASSOC(X))
-- induction hypothesis,
-- i.e. @assoc(l1,L2:List,L3:List) = true
  eq (l1 @ L2:List) @ L3:List = l1 @ (L2 @ L3) .
-- arbitrary element e
  op e : -> Elt }
--  check whether "@assoc(e | l1,l2,l3)" is deducible
--  at "@ASSOC-iStep"
--> [01] sub-goal 1 for the goal [0]
red in @ASSOC-iStep : @assoc(e | l1,l2,l3) .
--> returns "true"
--> QED
```

$$\frac{\texttt{@ASSOC} \models \texttt{@assoc(nil,l2,l3)} \qquad \begin{array}{l}\texttt{@ASSOC} \models \\ \big(\texttt{@assoc(l1,L2:List,L3:List)} \\ \quad \Rightarrow \texttt{@assoc(e|l1,l2,l3)}\big)\end{array}}{\texttt{@ASSOC} \models \texttt{@assoc(l1,l2,l3)}}$$

Focuses to constructor-based order-sorted equational specifications on which our proof score method has been mainly developed. For defining models and satisfaction relations the following concepts are going to be defined.

- a class $\mathbb{S}ign$ of **signatures**,
- for each signature $\Sigma \in \mathbb{S}ign$ a class $\mathrm{MOD}(\Sigma)$ 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 practically a finite collection of sentences (equations) E for the some signature Σ, and defined by a pair of the signature and the collection of sentences. That is, $SP = (\Sigma, E)$.

- The **denotation** of a specification is a class of all the models (i.e. possible implementations) of the specification.
- A specification is **basic** or **structured**.
- The **loose denotation** of a specification is the class $\mathrm{MOD}(SP)$ of all models of $\mathbb{S}ig(SP)$ which satisfy all sentences in SP.
- The **tight denotation** consists only of the **initial model** 0_{SP} in $\mathrm{MOD}(SP)$, i.e., for each other model $M \in \mathrm{MOD}(SP)$ of SP there exists a unique model morphism $0_{SP} \to M$.
- CafeOBJ supports the distinction between loose and tight denotations by special keywords, **mod!** for tight semantics, and **mod*** for loose semantics.

Signatures are formed by a set of **sorts** and **operators** on the set of sorts.

- A **sort** is a name for entities of the same type. Semantically, a sort denotes the set of entities of that type (sort).
- CafeOBJ supports subtyping via the **subsort** construct which specifies an inclusion between two sets.
- `s < s'` means that the set of sort s is subset of or equal to the set s'. `s1 s2 < s` is an abbreviation of "`s1 < s` and `s2 < s`"
- the set of sorts S is understood as the partial ordered set (POSET) $(S, \leq)$
- Given a poset $(S, \leq)$, 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 $(S, \leq)$.

- An **operator** (or function) f on a set of sorts S is denoted as $f : w \to s$ where $w \in S^*$ is its **arity** and $s \in S$ is its **sort** (sometimes called **co-arity**) of the operator.
- 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, $\exists f \in F_{ws} \cup F_{w's'}$ for different ws and $w's'$.
- CafeOBJ has a built-in module `BOOL` with the sort `Bool`, and an operator with co-arity of `Bool` is called **predicate**.

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

  $(w \equiv_{\leq} w' \Rightarrow s \equiv_{\leq} s')$ for any $f \in F_{ws} \cap F_{w's'}$.

  Where $w \equiv_{\leq} w'$ means that (1) w and w' are of the same length and (2) any element of w is in the same connected component with corresponding element of w'. Notice that $[] \equiv_{\leq} []$ for the empty arity $[]$.

Example

In CafeOBJ notation,

```
{ [Bool Nat]
    op 0 : -> Bool
    op 0 : -> Nat }
```

defines a non-sensible signature, and `0` can not 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 which belongs to `Nat` and `EvenInt`, but it has no minimal parse.

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

- A sort $s \in S$ is **constrained** if
  1. there exists a operator $f \in F^c_{ws}$ with the result sort s, or
  2. there exists a constrained sort s' such that $s' \leq s$.
- S^c : the set of constrained sorts
  $S^l \stackrel{\mathrm{def}}{=} S - S^c$: the set of loose sorts

Example

The module LIST determines the constructor-based order-sorted signature $\mathbb{S}ig(\texttt{LIST}) = (\mathrm{S}, \leq, \mathrm{F}, \mathrm{F}^{\mathrm{c}})$ as follows.

$\mathsf{S} = \{\texttt{Bool}, \texttt{Elt}, \texttt{Nil}, \texttt{NnList}, \texttt{List}\}$

$< = \{(\texttt{Nil List}), (\texttt{NnList List})\}$

$\mathsf{F} = \{\mathrm{F}_{ws}\}_{w \in \mathrm{S}^*, s \in \mathrm{S}}$

where $\mathrm{F}_{\texttt{Bool}} = \{\texttt{true}, \texttt{false}\}$, $\mathrm{F}_{\texttt{Nil}} = \{\texttt{nil}\}$,

$\mathrm{F}_{\texttt{Elt List NnList}} = \{\_|\_\}$, $\mathrm{F}_{\texttt{List List Bool}} = \{\_=\_\}$,

$\mathrm{F}_{ws} = \{\}$ otherwise.

$\mathrm{F}^{c} = \{\mathrm{F}^{c}_{ws}\}_{w \in \mathrm{S}^*, s \in \mathrm{S}}$

where $\mathrm{F}^{\mathrm{c}}{}_{\texttt{Nil}} = \{\texttt{nil}\}$, $\mathrm{F}^{\mathrm{c}}{}_{\texttt{Elt List NnList}} = \{\_|\_\}$,

$\mathrm{F}^{\mathrm{c}}{}_{ws} = \{\}$ otherwise.

$\mathrm{S}^{c} = \{\texttt{Bool}, \texttt{Nil}, \texttt{NnList}, \texttt{List}\}$. $\mathrm{S}^{l} = \{\texttt{Elt}\}$.

Order-sorted algebra

A $(S, \leq, F)$-**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 $f \in F_{s_1 \dots s_n s}$ as an operator

  $M_f : M_{s_1} \times \cdots \times M_{s_n} \to M_s$

  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 $M_{f:w \to s}(\overline{a}) = M_{f:w' \to s'}(\overline{a})$.

Order-sorted algebra

A $(S, \leq, F)$-**algebra** M consists of:

- Order-sorted family of carrier sets $\{M_s\}_{s \in S}$ satisfying ($s \leq s' \Rightarrow M_s \subseteq M_{s'}$), and
- Set of operators

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

such that any two operators of the same name return the same value if applied to the same argument.

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

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

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

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. $\Sigma(X)$-**term** is defined recursively as follows. Notice that sensibility makes the definition consistent.

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

Terms and term algebras

- $T_\Sigma(X) \stackrel{\text{def}}{=} \{T_\Sigma(X)_s | s \in S\}$
- $\Sigma(\{\})$-term is called Σ-term (or **ground**-term).
- $T_\Sigma \stackrel{\text{def}}{=} T_\Sigma(\{\})$
  The S-sorted set of Σ-ground-terms.
- The $T_\Sigma(X)$ or T_Σ can be organized as a Σ-algebra in the obvious way by using the above inductive definition of Σ-terms.
- CafeOBJ is a language for modeling systems in Σ-algebras.

FT T_Σ has the following **initiality** property:
Let Σ be an $(S, \leq, F)$-signature. For any Σ-algebras M there exists a unique Σ-algebra-morphism $T_\Sigma \to M$.

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

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

Sentences of equational specifications are equations.

- Given a signature Σ, an equational atom is $t = t'$, where $t, t' \in T_\Sigma(X)$ for some sorted set of variables X.
- A conditional Σ-**equation** is

$$(\forall X)\ t = t' \ \ \textbf{if} \ \ C$$

  where C is a set of equational atoms and is the **condition** of the equation.
- When the condition is empty it is called **unconditional** equation, and is written as

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

Valuations and term interpretation

- Valuations assign values to variables, in other words they represent instantiations of the variables with values from a given model. Let Σ be $(S, \leq, F)$-signature. Given Σ-model M and an S-sorted set X of variables, a valuation $\theta : X \to M$ consists of an S-sorted family of maps $\{\theta_s : X_s \to M_s\}_{s \in S}$.
- Each Σ(X)-term t can be interpreted as a value $\theta(t)$ in the model M for each valuation $\theta : X \to M$ in the following inductive manner:
  - M_f if t is a constant f,
  - $\theta(x)$ if t is a variable x,
  - $M_f(\theta(t_1), \ldots, \theta(t_n))$ if t is of the form $f(t_1, \ldots, t_n)$ for some $f \in F_{s_1 \ldots s_n s}$ and terms t_i of sort s_i.

Equation satisfaction

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

$$M \models_\Sigma ((\forall X)\ t = t' \ \ \textbf{if} \ \ C)$$

  iff $\theta(t) = \theta(t')$ whenever $\theta(C)$ for all valuations $\theta : X \to M$. Where $\theta(C)$ means $(\forall\, t_c = t'_c \in C)\, \theta(t_c) = \theta(t'_c)$. Notice that $\theta(\{\})$ holds for any valuation θ.
- An equation is satisfied by an algebra iff all possible ways to assign values to variables evaluate both sides of the equation as the same value, with proviso that the condition C is satisfied.

$\mathrm{M}\scriptstyle{\mathrm{OD}}$$(SP)$, $SP \models e$, $SP \models E$

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

- A Σ-algebra M is a model of a specification $SP = (\Sigma, E)$ iff $((\forall\, e \in E)\, M \models_\Sigma e)$.
- $\mathrm{MOD}(SP)$ is the set of all models that satisfy SP.
- An Σ-equation e is defined to be **satisfied** by a specification SP, denoted as $SP \models e$, iff $((\forall\, M \in \mathrm{MOD}(SP))\, M \models e)$.
- A set of Σ-equations E is defined to be **satisfied** by a specification SP, denoted as $SP \models E$, iff $((\forall\, e \in E)\, SP \models e)$.

A **congruence** $\equiv$ on an $(S, \leq, F)$-algebra M is an S-sorted equivalence on M (i.e., an equivalence $\equiv_s$ on M_s for each sort $s \in S$) such that

- if $a_i \equiv_{s_i} a'_i$ for $i \in \{1, \ldots, n\}$ then $M_f(a_1, \ldots, a_n) \equiv_s M_f(a'_1, \ldots, a'_n)$ for all $f \in F_{s_1 \ldots s_n s}$ and for all ranks $s_1 \ldots s_n s$.

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

- for each $s \in S$ let $(T_{\Sigma,E})_s$ be the set of equivalence classes of Σ-terms in T_Σ under the congruence $\equiv^E$ defined as $t \equiv^E t'$ iff $(\Sigma, E) \models (\forall\{\})\, t = t'$.
- each operation $f \in F_{s_1 \ldots s_n s}$ is interpreted as $(T_{\Sigma,E})_f(t_1/{\equiv^E}, \ldots, t_n/{\equiv^E}) = f(t_1, \ldots, t_n)/{\equiv^E}$ for all $t_i \in (T_\Sigma)_{s_i}$ $(i \in \{1, \ldots, n\})$ by using the property of $\equiv^E$ as congruence on T_Σ.

The initial algebra of order-sorted algebras

$T_{\Sigma,E}$ has the following **initiality** property, and is the model giving the tight denotation of the equational specification (Σ, E).

FT Let (Σ, E) be an equational specification of order-sorted signature Σ which does not contain constructor declarations. For any Σ-algebra M satisfying all equations in E, there exists a unique Σ-algebra-morphism $T_{\Sigma,E} \to M$.

Sufficiently completeness

Let SP be a constructor-based order-sorted specification with the signature $(S, \leq, F, F^c)$, and

S^c be the set of constrained sorts, and

S^l be the set of loose sorts, and

$F^{S^c} \stackrel{\text{def}}{=} \{f : w \to s \mid f \in F, s \in S^c\}$, and

$\Sigma^{S^c} \stackrel{\text{def}}{=} (S, \leq, F^{S^c})$, and

$\Sigma^c \stackrel{\text{def}}{=} (S, \leq, F^c)$, and

Y be any S^l sorted set of variables.

A specification SP is defined to be **sufficiently complete** if for any term $t \in T_{\Sigma^{S^c}}(Y)$ there exits a term $t' \in T_{\Sigma^c}(Y)$ such that $SP \models (\forall Y)\, t = t'$.

The initial algebra of constractor-based order-sorted algebras

Sufficiently completeness is a sufficient condition for the existence of the initial algebra of constructor-based order-sorted algebras.

FT Let $SP = (\Sigma, E)$ be a constructor-based order-sorted specification with the signature $\Sigma = (S, \leq, F, F^c)$. If the specification SP is sufficiently complete, for any Σ-algebra M satisfying all equations in E, there exists a unique Σ-algebra-morphism $T_{\Sigma,E} \to M$.

We consider the following four specification building operations of **BS**, **SU**, **PR**, **IN** for constructing a new specification from old ones.

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

(SU) A new specification $SP_1 \cup SP_2$ is built by making sum of two specifications SP_1 and SP_2 with the same signature Σ. That is,

$$\mathbb{S}ig(SP_1 \cup SP_2) \stackrel{\text{def}}{=} \mathbb{S}ig(SP_1) = \mathbb{S}ig(SP_2) = \Sigma,$$
$$\textsc{Mod}(SP_1 \cup SP_2) \stackrel{\text{def}}{=} \textsc{Mod}(SP_1) \cap \textsc{Mod}(SP_2).$$

(PR) A new specification $\mathtt{PR}(SP, \Sigma')$ is built by protecting a specification SP and add a new part of signature Σ'. That is,

$$\mathbb{S}ig(\mathtt{PR}(SP, \Sigma') \stackrel{\text{def}}{=} \mathbb{S}ig(SP) \cup \Sigma',$$

$$\mathrm{MOD}(\mathtt{PR}(SP, \Sigma')) \stackrel{\text{def}}{=}$$

$$\{M \in \mathrm{MOD}((\mathbb{S}ig(SP) \cup \Sigma', \{\})) \mid M{\restriction}_{\Sigma} \in \mathrm{MOD}(SP)\},$$

where $M{\restriction}_{\mathbb{S}ig(SP)}$ is $\mathbb{S}ig(SP)$ part of the model M.

(IN) A new specification $SP\,!$ is built by declaring the tight denotation. That is,

$$\mathbb{S}ig(SP\,!) \stackrel{\text{def}}{=} \mathbb{S}ig(SP), \text{ and}$$

$$\mathrm{MOD}(SP\,!) \stackrel{\text{def}}{=}$$

$$\begin{cases} \{0_{SP}\} & \text{if the initial algebra of } \mathrm{MOD}(SP) \text{ exists} \\ \{\} & \text{otherwise.} \end{cases}$$

Equation calculus is a syntactic definition of equational deduction with respect to a fixed SP. The equational calculus for an equational specification $((S, \leq, F), E)$ is defined by the following rules. Notice that this calculus is for deducing an unconditional equation.

$$\text{[reflexivity]} \quad \frac{}{(\forall X)\, t = t} \qquad \text{[symmetry]} \quad \frac{(\forall X)\, t = t'}{(\forall X)\, t' = t}$$

$$\text{[transitivity]} \quad \frac{(\forall X)\, t = t' \qquad (\forall X)\, t' = t''}{(\forall X)\, t = t''}$$

$$\text{[congruence]} \quad \frac{(\forall X)\, t_i = t'_i \quad \text{for all } i \in \{1, \ldots, n\}}{(\forall X)\, f(t_1, \ldots, t_n) = f(t'_1, \ldots, t'_n)}$$

for all operations $f \in F_{s_1 \ldots s_n s}$, and t_i of sort s_i for all $i \in \{1, \ldots, n\}$.

$$\text{[instantiation]} \quad \frac{(\forall X)\, \theta(t_i) = \theta(t'_i) \;\; \text{for all } t_i{=}t'_i \in C}{(\forall X)\, \theta(t) = \theta(t')}$$

for any conditional equation $((\forall Y)\, t = t' \;\; \textbf{if} \;\; C)$ in E and any valuation $\theta : Y \rightarrow T_\Sigma(X)$.

Let $SP \vdash^{\mathrm{eq}} e$ denote that a unconditional equation e $(= (\forall X)\, t = t')$ is deducible by the equation calculus with respect to SP. $SP \vdash^{\mathrm{eq}} e$ is called an **equation entailment**.

FT With respect to an order sorted equational specification SP, the equation calculus is **sound** in the sense that $((SP \vdash^{\mathrm{eq}} e)$ implies $SP \models e)$ holds.
If the specification SP does not contain constructor declarations (i.e. $SP = ((S, \leq, F, \{\}), E))$, the equation calculus is **complete** with respect to the denotational semantics in the sense that $(SP \models e$ implies $SP \vdash^{\mathrm{eq}} e)$ holds.

- The reduction command "`red in` SP `:` t `.`" (for a ground term t) of CafeOBJ applies all the equations of SP as rewriting rules from left to right as much as possible and gets a normal form of t.
- For interpreting equations as rewriting rules, the following syntactic condition is used in CafeOBJ.

$$var(t') \subseteq var(t) \quad \text{and} \quad var(C) \subseteq var(t)$$

  where $var(t)$ is the set of variables occurring in the term t and $var(C)$ is the set of variables occurring in a term that constitutes some equation in C.

- Let $SP \vdash^{\mathrm{eq}} t \,\texttt{<->}_{\texttt{rd}}\, t'$ denote that the CafeOBJ reduction command "`red in` SP `:` $t = t'$ `.`" returns `true`. Because of the honesty of CafeOBJ reduction to the equation calculus, the following holds.

$$[\text{cafeRed}] \quad \frac{SP \vdash^{\mathrm{eq}} t \,\texttt{<->}_{\texttt{rd}}\, t'}{SP \vdash^{\mathrm{eq}} (\forall\{\})\, t = t'}$$

This rule is the base for constructions of proof trees for the verifications with proof scores.

```
-- using built-in BOOL
set include BOOL on
--> a set of elements with a void element
mod* TRIVvo {[Elt] op vo : -> Elt}
--> parametrized 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)) . }
```

```
mod* @ASSOCvo(X :: TRIVvo){pr(APPENDvo-ASSOC(X))
-- for arbitrary lists l1 l2 l3
ops l1 l2 l3 : -> List }
--> [0] the goal
-- check whether "@assoc(l1,l2,l3)" is deducible
-- at "@ASSOC"
red in @ASSOCvo : @assoc(l1,l2,l3) .
--> does not return "true"
```

```
**> decide to use induction w.r.t.
**> the first argument l1 of "@assoc(l1,l2,l3)"
--> Induction base
mod* @ASSOCvo-iBase(X :: TRIVvo){pr(@ASSOCvo(X))}
--> [00] sub-goal 0 for the goal [0]
-- check whether "@assoc(nil,l2,l3)" is deducible
-- at "@ASSOC-iBase"
red in @ASSOCvo-iBase : @assoc(nil,l2:List,l3:List) .
--> returns "true"
```

```
--> Induction step
mod* @ASSOCvo-iStep(X :: TRIVvo){pr(@ASSOCvo(X))
-- induction hypothesis,
-- i.e. @assoc(l1,L2:List,L3:List) = true
  eq (l1 @ L2:List) @ L3:List = l1 @ (L2 @ L3) .
-- for arbitrary element e
  op e : -> Elt.X . }
--> [01] sub-goal 1 for the goal [0]
-- check whether "@assoc(e | l1,l2,l3)" is deducible
-- at "@ASSOC-iStep"
red in @ASSOCvo-iStep : @assoc(e | l1,l2,l3) .
--> does not return "true"
```

```
**> decide to do case splitting
**> using the predicate (e = vo)
--> case of ((e = vo) = true) i.e. (e = vo)
mod* @ASSOCvo-iStep-c0(X :: TRIVvo)
     {pr(@ASSOCvo-iStep(X))
eq e = vo .}
--> [010] sub-goal 0 for sub-goal [01]
--  check whether "@assoc(e | l1,l2,l3)" is deducible
--  at @ASSOC-iStep-c0
red in @ASSOCvo-iStep-c0 : @assoc(e | l1,l2,l3) .
--> returns "true"
```

```
--> case of ((e = vo) = false)
mod* @ASSOCvo-iStep-c1(X :: TRIVvo)
     {pr(@ASSOCvo-iStep(X))
eq (e = vo) = false .}
--> [011] sub-goal 1 for sub-goal [01]
--  check whether "@assoc(e | l1,l2,l3)" is deducible
--  at @ASSOC-iStep-c1
red in @ASSOCvo-iStep-c1 : @assoc(e | l1,l2,l3) .
--> returns "true"
--> QED
```

$$\frac{\texttt{@ASSOCvo-iStep-c0} \models \texttt{@assoc(e | l1,l2,l3)} \qquad \texttt{@ASSOCvo-iStep-c1} \models \texttt{@assoc(e | l1,l2,l3)}}{\texttt{@ASSOCvo-iStep} \models \texttt{@assoc(e | l1,l2,l3)}}$$

Overall Proof Tree

$$
\dfrac{
\dfrac{\texttt{@ASSOC} \vdash^{\mathrm{eq}} \texttt{@assoc(nil,l2,l3)} \; \texttt{<->}_{\texttt{rd}} \, \texttt{true}}{\texttt{@ASSOC} \models \texttt{@assoc(nil,l2,l3)}}
\qquad
\dfrac{
\dfrac{\texttt{@ASSOCvo-iStep-c0} \vdash^{\mathrm{eq}} \texttt{@assoc(e | l1,l2,l3)} \; \texttt{<->}_{\texttt{rd}} \, \texttt{true}}{\texttt{@ASSOCvo-iStep-c0} \models \texttt{@assoc(e | l1,l2,l3)}}
\quad
\dfrac{\texttt{@ASSOCvo-iStep-c1} \vdash^{\mathrm{eq}} \texttt{@assoc(e | l1,l2,l3)} \; \texttt{<->}_{\texttt{rd}} \, \texttt{true}}{\texttt{@ASSOCvo-iStep-c1} \models \texttt{@assoc(e | l1,l2,l3)}}
}{\texttt{@ASSOCvo-iStep} \models \texttt{@assoc(e | l1,l2,l3)}}
}{\texttt{@ASSOC} \models \texttt{@assoc(l1,l2,l3)}}
$$

$$[\text{initMod}] \quad \frac{0_{SP} \models_{\mathbb{S}ig(SP)} e}{SP\,! \vdash^{\mathrm{sp}} e}$$

$$[\text{cafeRed}] \quad \frac{SP \vdash^{\mathrm{eq}} t \texttt{<->}_{\texttt{rd}} t'}{SP \vdash^{\mathrm{eq}} (\forall\{\})\, t = t'}$$

$$[\text{eqToSp}] \quad \frac{SP \vdash^{\mathrm{eq}} e}{SP \vdash^{\mathrm{sp}} e}$$

$$[\text{axiom}] \quad \frac{}{(\Sigma, E \cup \{e\}) \vdash^{\text{sp}} e} \qquad [\text{protect}] \quad \frac{SP \vdash^{\text{sp}} e}{\mathrm{PR}(SP, \Sigma') \vdash^{\text{sp}} e}$$

$$[\text{lemma}] \quad \frac{SP \vdash^{\text{sp}} \{e_1, \ldots, e_n\} \qquad SP \cup (\mathbb{S}ig(SP), \{e_1, \ldots, e_n\}) \vdash^{\text{sp}} e}{SP \vdash^{\text{sp}} e}$$

Here $SP \vdash^{\text{sp}} \{e_1, \ldots, e_n\} \stackrel{\text{def}}{=} \{SP \vdash^{\text{sp}} e_i \mid e_i \in \{e_1, \ldots, e_n\}\}$.

$$[\text{sum}] \quad \frac{SP_1 \vdash^{\text{sp}} e}{SP_1 \cup SP_2 \vdash^{\text{sp}} e} \qquad [\text{union}] \quad \frac{SP \vdash^{\text{sp}} E_1 \quad SP \vdash^{\text{sp}} E_2}{SP \vdash^{\text{sp}} E_1 \cup E_2}$$

Here E_1 and E_2 is sets of equations; a equation e can be understood as a singleton set of the equation $\{e\}$.

$$[\mathrm{thConst1}] \quad \frac{SP \vdash^{\mathrm{sp}} (\forall Y)\,\varepsilon}{\mathtt{PR}(SP, Y) \vdash^{\mathrm{sp}} (\forall \{\})\,\varepsilon}$$

$$[\mathrm{thConst2}] \quad \frac{\mathtt{PR}(SP, Y) \vdash^{\mathrm{sp}} (\forall \{\})\,\varepsilon}{SP \vdash^{\mathrm{sp}} (\forall Y)\,\varepsilon}$$

$$[\text{condEq1}] \quad \frac{(\Sigma, E) \vdash^{\text{sp}} (\forall\{\})\ t = t' \ \ \textbf{if}\ \ \{t_1 = t'_1, \ldots, t_n = t'_n\}}{(\Sigma, E \cup \{(\forall\{\})\ t_1 = t'_1, \ldots, (\forall\{\})\ t_n = t'_n\}) \vdash^{\text{sp}} (\forall\{\})\ t = t'}$$

$$[\text{condEq2}] \quad \frac{(\Sigma, E \cup \{(\forall\{\})\ t_1 = t'_1, \ldots, (\forall\{\})\ t_n = t'_n\}) \vdash^{\text{sp}} (\forall\{\})\ t = t'}{(\Sigma, E) \vdash^{\text{sp}} (\forall\{\})\ t = t' \ \ \textbf{if}\ \ \{t_1 = t'_1, \ldots, t_n = t'_n\}}$$

$$\text{[imp1]} \quad \frac{SP \vdash^{\mathrm{sp}} (\forall\{\})\ t = t' \ \ \textbf{if} \ \ \{t_1 = t_1', \ldots, t_n = t_n'\}}{SP \vdash^{\mathrm{sp}} (\forall\{\})\,((t_1 = t_1' \ \mathtt{and}, \ldots, \mathtt{and}\ t_n = t_n')\ \mathtt{implies}\ t = t') = \mathtt{true}}$$

$$\text{[imp2]} \quad \frac{SP \vdash^{\mathrm{sp}} (\forall\{\})\,((t_1 = t_1' \ \mathtt{and}, \ldots, \mathtt{and}\ t_n = t_n')\ \mathtt{implies}\ t = t') = \mathtt{true}}{SP \vdash^{\mathrm{sp}} (\forall\{\})\ t = t' \ \ \textbf{if} \ \ \{t_1 = t_1', \ldots, t_n = t_n'\}}$$

$$[\text{conAbst}] \quad \frac{\{SP \vdash^{\mathrm{sp}} (\forall Y)\, \theta(\varepsilon) \mid \theta : X \to T_{\Sigma^c}(Y), Y : \text{finite}\}}{SP \vdash^{\mathrm{sp}} (\forall X)\, \varepsilon}$$

$$[\text{conInd}] \quad \frac{\begin{array}{c} SP' \stackrel{\text{def}}{=} \mathtt{PR}(SP, \{\{x\}_s\}) \cup \{(\forall\{\})\, \varepsilon\} \\ \{SP' \vdash^{\mathrm{sp}} (\forall Z^f)\, \varepsilon[x \leftarrow f(z_1, .., z_{i-1}, x, z_{i+1}, .., z_n)] \mid f \in F^c_{*s}\} \end{array}}{SP \vdash^{\mathrm{sp}} (\forall\{\{x\}_s\})\, \varepsilon}$$

$$[\text{caseSplit}] \quad \frac{\{\mathtt{PR}(SP, Y) \cup \{u = t\} \vdash^{\mathrm{sp}} e \;\mid\; t \in T_{\Sigma^c}(Y)_{s_c}, Y : \text{finite}\}}{SP \vdash^{\mathrm{sp}} e}$$

$$[\text{caseSplitBool}] \quad \frac{SP \cup \{u = \mathtt{true}\} \vdash^{\mathrm{sp}} e \qquad SP \cup \{u = \mathtt{false}\} \vdash^{\mathrm{sp}} e}{SP \vdash^{\mathrm{sp}} e}$$

Nodes, trees, roots, and sub-trees are defined as follows.

(T1) An entailment "$SP \vdash^{\mathrm{sp}} e$", "$SP \vdash^{\mathrm{eq}} e$", or "$SP \vdash^{\mathrm{eq}} t \texttt{<->}_{\mathtt{rd}} t'$" is a node which is called sp-node, eq-node, or rd-node respectively. A node n is a tree, and n is called the root of the tree.

(T2) If n is a node and $t_1, \ldots, t_i$ for $i \in \{0, 1, \ldots\}$ are trees, $(\{t_1, \ldots, t_i\}, n)$ is a tree. n is called the root of the tree, and $t_1, \ldots, t_i$ are called sub-trees of the tree or the node n. Sub-tree is transitive relation and if t_a is a sub-tree of t_b and t_b is sub-tree of t_c then t_a is a sub-tree of t_c. If $i = 0$ then $(\{\}, n)$ is a tree with empty sub-trees. Whereas, a node is a tree with no sub-trees.

Based on the proof rules in the specification calculus, **p-trees (proof trees)** are defined as follows.

(T3) A tree with empty sub-trees, an eq-node, or a rd-node is a p-tree. Let $(\{t_1, \ldots, t_i\}, n)$ be a tree, and $n_1, \ldots, n_i$ be the roots of the sub-trees $t_1, \ldots, t_i$ respectively. The tree $(\{t_1, \ldots, t_i\}, n)$ is a p-tree if (1) $t_1, \ldots, t_i$ are p-trees, and (2) $\frac{n_1, \ldots, n_i}{n}$ is an instance of one of the proof rules of the specification calculus.

(T4) A sub-tree of a tree is a leaf if (1) it is a node, or (2) it is a tree with empty sub-trees. A p-tree is also defined to be a tree such that any of whose leafs is (1) a tree with empty sub-trees, (2) an eq-node, or (3) a rd-node.

- A node leaf is a leaf that is a node. A set of node leafs of a p-tree is called a **proof score** of the p-tree if any eq-node in the set is of the form $SP \vdash^{\mathrm{eq}} (\forall\{\})\, t = t'$. Notice that the validity of this kind of equation entailment can be checked by CafeOBJ system to execute the reduction command of "$\texttt{red in } SP \texttt{ : } t = t' \texttt{ .}$".
- If any leaf of a p-tree is either a tree with empty sub-trees or a rd-node, the p-tree is called **effective**. An **effective proof score** is a proof score of an effective p-tree. Notice that an effective proof score consists only of rd-nodes (i.e. entailments of the form "$SP \vdash^{\mathrm{eq}} t \texttt{<->}_{\texttt{rd}}\, t'$") whoes validity are proved by executing CafeOBJ reduction commands.

- Given a predicate p about a specification SP, if we can construct an effective p-tree whose root is the entailment $SP \vdash^{\mathrm{sp}} (p = \texttt{true})$ then the satisfaction assertion $SP \models (p = \texttt{true})$ is proved to hold.

$$
\dfrac{
\dfrac{\texttt{@ASSOC} \vdash^{\mathrm{eq}} \texttt{@assoc(nil,l2,l3)} \; \texttt{<->}_{\texttt{rd}}\, \texttt{true}}{\texttt{@ASSOC} \vdash^{\mathrm{sp}} \texttt{@assoc(nil,l2,l3)}}
\qquad
\dfrac{
\dfrac{\texttt{@ASSOCvo-iStep-c0} \vdash^{\mathrm{eq}} \texttt{@assoc(e | l1,l2,l3)} \; \texttt{<->}_{\texttt{rd}}\, \texttt{true}}{\texttt{@ASSOCvo-iStep-c0} \vdash^{\mathrm{sp}} \texttt{@assoc(e | l1,l2,l3)}}
\quad
\dfrac{\texttt{@ASSOCvo-iStep-c1} \vdash^{\mathrm{eq}} \texttt{@assoc(e | l1,l2,l3)} \; \texttt{<->}_{\texttt{rd}}\, \texttt{true}}{\texttt{@ASSOCvo-iStep-c1} \vdash^{\mathrm{sp}} \texttt{@assoc(e | l1,l2,l3)}}
}{\texttt{@ASSOCvo-iStep} \models \texttt{@assoc(e | l1,l2,l3)}}
}{\texttt{@ASSOC} \vdash^{\mathrm{sp}} \texttt{@assoc(l1,l2,l3)}}
$$

Let $SP \vdash e$ denote that a p-tree with the root of $SP \vdash^{\mathrm{sp}} e$ can be constructed.

PR [soundness] $SP \vdash E$ implies $SP \models E$.

We need **sufficiently completeness** for describing the converse implication precisely.

TH [quasi-completeness] $SP \models E$ implies $SP \vdash E$ if

- SP is formed by applying the three specification building operators of **BS**, **SU** and **PR**,
- SP is sufficiently complete.

- CafeOBJ is a language for systems specification based on algebraic abstract types, and has a high potential to describe **specifications in an appropriate abstraction level**.
- Automated parts of verification are done solely by rewriting (or reduction) of CafeOBJ language system which is honest to equational deduction. And the 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 which satisfy specifications**. The specification calculus is based on this semantics and formalize the verification procedures at the level of goals expressed as satisfaction assertions $SP \models e$.

- To develop the theory or method to guarantee that every specification appearing during the specification calculus is **terminating, confluent, and/or sufficiently complete** as a TRS.
- **Constructions of p-trees and proof scores** themselves can be specified and analysed, and/or verified in CafeOBJ/Maude based on the specification calculus. It can lead to semi-automatic construction of p-trees and proof scores, and is a challenging research topic in the future.