On Equality Predicates in Algebraic Specification Languages

Nakamura Masaki and Futatsugi Kokichi

School of Information Science, Japan Advanced Institute of Science and Technology

Abstract. The execution of OBJ algebraic specification languages is based on the term rewriting system (TRS), which is an efficient theory to perform equational reasoning. We focus on the equality predicate implemented in OBJ languages. The equality predicate is used to test the equality of given terms by TRS. Unfortunately, it is well known that the current execution engine of OBJ languages with the equality predicate is not sound. To solve this problem, we define a modular term rewriting system (MTRS), which is suitable for the module system of OBJ languages, and propose a new equality predicate based on MTRS.

1 Introduction

We propose a new equality predicate for algebraic specification languages that support a module system and a rewrite engine. The principals of the module system for algebraic specifications were first realized in the Clear language [7] and have been inherited by OBJ languages [9,12,3,5,2]. The theory of the CafeOBJ module system updates the original concepts of Clear or other OBJ languages to a more sophisticated situation [8], which helps describe the specifications of a large and complex system by using several types of module imports, built-in modules and the loose and tight denotation for each module. The execution of OBJ languages is based on the term rewriting system (TRS) [14,15], which is a useful notion for realizing equational reasoning, the most basic building block of the verification of OBJ specifications. Using a rewriting engine based on TRS, we obtain a powerful semi-automatic verification system. Although OBJ languages support a sophisticated module system for specification description, the specification verification, however, does not benefit greatly from the module system. Actually, in the current implementation of CafeOBJ, the rewrite engine treats all equations equally. OBJ languages support the equality predicate, which is a special operation symbol used to test the equality of terms. However, the use of the equality predicate in a specification makes its verification unsound [11].

We present an example of CafeOBJ modules to show that the current equality predicate is problematic. The following specification `Z` denotes the set of integers. The constant operator `0` stands for 0, and the unary operators `s` and `p` are the successor and predecessor functions. The equations mean that the successor of the predecessor, and the predecessor of the successor, of an integer is the integer itself, where `X` is a variable that denotes an arbitrary integer.

C.B. Jones, Z. Liu, J. Woodcock (Eds.): ICTAC 2007, LNCS 4711, pp. 381–395, 2007.

```
mod! Z{ [Zero < Int]
  op 0 : -> Zero
  ops s p : Int -> Int
  eq s(p(X:Int)) = X .
  eq p(s(X:Int)) = X .}
```

The equality predicate `_==_` is used for checking the equality of terms, for example, `s(p(X)) == p(s(X))` means the predecessor of the successor is equivalent to the successor of the predecessor for any integer. The equational reasoning with the CafeOBJ system is performed as follows: the system first reduces the both sides of the equation, and returns `true` if the results are the same, and otherwise returns `false`. Here, terms are reduced according to the equations in the specification where the equations are regarded as left-to-right oriented rewrite rules. For the above equation, the both sides are reduced into `X`, and the CafeOBJ system returns `true`. The equality predicate has a problem when being inside the specification. The following specification tt ZERO specifies the predicate that tests whether an integer is zero or not.

```
mod! ZERO{ pr(Z)
  op zero : Int -> Bool
  eq zero(X:Int) = (X == 0) .}
```

The equation in the specification defines the predicate `zero(X)` as the result of `X == 0`. We next try to prove the equation `zero(s(X)) == false`. The left-hand side is reduced into `false` as follows: `zero(s(X))` → `s(X) == 0` → `false`, and the CafeOBJ system returns `true` for the above equation, however, it is not true in the case of `X` = `p(0)`. Thus, the current CafeOBJ system is unsound if the equality predicate is used inside a specification. In the following sections, we discuss the reason for the unsoundness problem. We propose a term rewriting system based on the module system, called the modular term rewriting system (MTRS), and define a new equality predicate by using the MTRS to solve the unsoundness problem.

In the next section, we briefly introduce CafeOBJ algebraic specification language. We focus on the module system. In Section 3, we propose a modular equational proof system (MEPS) for the module system of the CafeOBJ specification language. In Section 4, we propose MTRS. In Section 5, we discuss the problems of the existing equality predicate and propose a new equality predicate. In Section 6, we discuss applications of our research, and we present conclusions in Section 7.

2 Preliminaries

We introduce OBJ algebraic specification languages [8,9,12,13,16] with the notations and definitions from CafeOBJ [8]. To simplify the discussion, we treat only a subset of CafeOBJ, which does not include, for example, conditional equations, parameterized modules and transition rules. However, the present results can be applied straightforwardly to full CafeOBJ specifications.

2.1 Algebraic Specification

Let S be a set. An S-sorted set A is a family $\{A_s \mid s \in S\}$ of sets indexed by elements of S. We may write $a \in A$ if $a \in A_s$ for some $s \in S$. S^* is the set of sequences of elements of S. The empty sequence is denoted by ε. S^+ is the set of non-empty sequences, i.e., $S^+ = S^* \setminus \{\varepsilon\}$. An (order-sorted) signature $(S, \leq, \Sigma)$ is a triple of a set S of sorts, a partial order $\leq \subseteq S \times S$, and S^+-sorted set Σ of operation symbols, where for $f \in \Sigma_{w\,s}$ ($w \in S^*$ and $s \in S$), $w\,s$ is referred to as the rank of f. When $w = \varepsilon$, f is referred to as a constant symbol. $(S, \leq, \Sigma)$ is occasionally abbreviated as Σ. Let $(S, \leq, \Sigma)$ be a signature and X an S-sorted set of variables. An S-sorted set $T(\Sigma, X)$ (abbr. T) of terms is defined as the smallest set satisfying the following: (1) $\Sigma_s \subseteq T_s$ for any $s \in S$, (2) $X_s \subseteq T_s$ for any $s \in S$, (3) $f(t_1, \ldots, t_n) \in T_s$ if $f \in \Sigma_{s_1 \cdots s_n\, s}$ and $t_i \in T_{s_i}$ $(i = 1, \ldots, n)$. (4) $T_{s'} \subseteq T_s$ if $s' \leq s$. An equation is denoted by $(\forall X)t = t'$, where X is a set of variables, $t, t' \in T(\Sigma, X)_s$ for some sort $s \in S$. We may omit $(\forall X)$ if no confusion exists. An (equational) specification is a pair of a signature and an axiom (a set of equations): $SP = (\Sigma, E)$. We write S_{SP}, $\leq_{SP}$, Σ_{SP}, and E_{SP} for the sets of the sorts, the partial order, the operation symbols, and the axiom of a specification SP. In addition, we also write $s \in SP\mathrm{C}f \in SP\mathrm{C}e \in SP\mathrm{C}t \in SP$ if $s \in S_{SP}$, $f \in \Sigma_{SP}$, $e \in E_{SP}$, $t \in T(\Sigma_{SP}, X)$, respectively. We may omit the signature of a specification and the specification is referred to as simply the axiom E.

Example 1. The following specification $SP_{\mathtt{Z}}$ denotes integers.

$$
\begin{aligned}
S_{SP_{\mathtt{Z}}} &= \{\mathtt{Zero}, \mathtt{Int}\} \\
\leq_{SP_{\mathtt{Z}}} &= \{(\mathtt{Zero}, \mathtt{Zero}), (\mathtt{Zero}, \mathtt{Int}), (\mathtt{Int}, \mathtt{Int})\} \\
\Sigma_{SP_{\mathtt{Z}}} &:= (\Sigma_{SP_{\mathtt{Z}}})_{\mathtt{Zero}} = \{\mathtt{0}\}, (\Sigma_{SP_{\mathtt{Z}}})_{\mathtt{Int\ Int}} = \{\mathtt{s}, \mathtt{p}\} \\
E_{SP_{\mathtt{Z}}} &= \{\mathtt{s(p(X))} = \mathtt{X},\ \mathtt{p(s(X))} = \mathtt{X}\}
\end{aligned}
$$

For a signature $(S, \leq, \Sigma)$, a Σ-algebra M is an algebra that consists of (1) an S-sorted carrier set M such that $M_s \subseteq M_{s'}$ if $s \leq s'$, (2) an element $M_c \in M_s$ for each $c \in \Sigma_s$, and (3) an operation (or a function) $M_f : M_{s_1} \times \cdots \times M_{s_n} \to M_s$ for each $f \in \Sigma_{s_1 \cdots s_n\, s}$. An assignment $a : X \to M$ is a map from an S-sorted variables set X to an S-sorted carrier set M such that $a(x) \in M_s$ if $x \in X_s$. By an assignment $a : X \to M$, a term $t \in T(\Sigma, X)$ can be interpreted as an element of M, denoted by $a(t)$, as follows: $a(t) = a(x)$ if $t = x \in X$, $a(t) = M_c$ if $t = c \in \Sigma$, and $a(t) = M_f(a(t_1), \ldots, a(t_n))$ if $t = f(t_1, \ldots, t_n)$. For a Σ-algebra M and an equation $e : (\forall X)t = t'$, we declare that M satisfies e, denoted by $M \models e$, iff $a(t) = a(t')$ for any assignment $a : X \to M$. For $SP = (\Sigma, E)$, a Σ-algebra that satisfies all equations in E is called an SP-algebra (or SP-model). We may omit SP- if no confusion exists. The set of all SP-algebras is denoted by $M(SP)$. For algebras A and B, a Σ-homomorphism $h : A \to B$ is an S-sorted homomorphism, which is a family of $\{h_s : A_s \to B_s\}_{s \in S}$, that satisfies the following: (M1) $h_s(x) = h_{s'}(x)$ for each $x \in A_s$ if $s \leq s'$. (M2) $h_s(A_f(a_1, \ldots, a_n))$ $= B_f(h_{s_1}(a_1), \ldots, h_{s_n}(a_n))$ if $f \in \Sigma_{s_1 \cdots s_n\, s}$ and $a_i \in A_{s_i} (i = 1, \ldots, n)$. The set of all Σ-homomorphisms is denoted by $H_{A,B}$. An initial SP-algebra is an SP-algebra I that has the unique Σ-homomorphism $h : I \to A$ for any SP-algebra

A. The set of all initial algebras is denoted by $IM(SP)$. Since initial algebras are isomorphic, we often identify one of the initial algebras with the set $IM(SP)$ of all initial algebras. Roughly speaking, an initial algebra I has the following properties: any element in I should be described in the signature of SP (no junk) and any equation in I should be deduced from the axiom of SP (no confusion).

Example 2. The following Z is an $SP_{\mathtt{Z}}$-algebra: $Z_{\mathtt{Zero}} = \{0\}$, $Z_{\mathtt{Int}} = \mathcal{Z}$, $Z_{\mathtt{0}} = 0$, $Z_{\mathtt{s}}(n) = n + 1$, $Z_{\mathtt{p}}(n) = n - 1$, where $\mathcal{Z}$ is the set of all integers. Z is initial ($Z \in IM(SP_{\mathtt{Z}})$). The real number algebra R with same interpretations and the Boolean algebra B with $B_{\mathtt{0}} = false$ $B_{\mathtt{s}} = B_{\mathtt{p}} = \neg$ can be $SP_{\mathtt{Z}}$-algebras; however, they are not initial. There is no term corresponding to the real number 3.1415, and although $B \models \mathtt{s(0)} = \mathtt{p(0)}$, it cannot be deduced from the axiom of `Z`.

2.2 CafeOBJ Algebraic Specification Language

We introduce CafeOBJ algebraic specifications language. Although in this paper we treat only CafeOBJ, other OBJ languages can be treated. The CafeOBJ specifications are described in a modular manner. The CafeOBJ module `Z` in Section 1 corresponds to $SP_{\mathtt{Z}}$ in Example 1: The declaration `[Zero < Int]` in `Z` corresponds to the sort set $\{\mathtt{Zero}, \mathtt{Int}\}$ and the reflexive transitive closure of the described order. An operation symbol is declared as `op f : A B -> C`, which stands for $\mathtt{f} \in \Sigma_{\mathtt{A\,B\,C}}$. By `ops`, we can declare two or more operation symbols with the same rank. $\Sigma_{\mathtt{A\,B\,C}}$ is the set of all operation symbols with the rank `A B C`. An equation is declared as `eq s(p(X:Int)) = X`, which stands for $(\forall X)\ \mathtt{s(p(X))} = \mathtt{X}$, where $X_{\mathtt{Int}} = \{\mathtt{X}\}$ and $X_s = \emptyset$ for any other $s \in S$. For a module MOD, the corresponding specification is denoted as SP_{MOD}. Denotations of CafeOBJ specifications represent the class of their algebras. In this section, we introduce specifications without explicit imports, called basic specifications (or basic modules). CafeOBJ basic specifications can be classified into specifications with a tight denotation and specifications with a loose denotation. A specification with a tight denotation is denoted as `mod!` and a loose specification is denoted as `mod*`. Hereinafter, each specification is assumed to have its denotation declaration and is referred to as $d(SP) = tight$ or $loose$. The denotation $[SP]$ of SP is defined as $IM(SP)$ if $d(SP) = tight$ and $M(SP)$ if $d(SP) = loose$. We refer to an element of $[SP]$ as a denotational model. We write $SP \models e$ if $M \models e$ for any $M \in [SP]$. Since `Z` is tight, $[SP_{\mathtt{Z}}] = IM(SP_{\mathtt{Z}})$. Thus, `Z` essentially denotes only the integer algebra Z. We next present an example of loose modules:

```
mod* FUN{ [Elt]
 op f : Elt -> Elt}
```

Since `FUN` is loose, $[SP_{\mathtt{FUN}}] = M(SP_{\mathtt{FUN}})$. Thus, `FUN` denotes all algebras, including at least one function on a set, e.g., they interpret f into, for example, the identity function on natural numbers, the sort function on arrays, a code refactoring, and a document editing.

The execution of CafeOBJ is based on the term rewriting system (TRS). The CafeOBJ system [3] is interactive. When starting CafeOBJ, we meet a prompt `CafeOBJ>`. After inputting a CafeOBJ module, e.g., `mod! TEST{...}`, by hand or through a file, we can select or open the module by `select TEST` or `open TEST`. A prompt will be changed to `TEST>` or `%TEST>`. We can then use the CafeOBJ reduction command `red`. Here, `red` takes a term constructed from the selected or opened module, and returns its normal form (in the default strategy) with respect to the TRS, the rewrite rules of which are left-to-right-oriented equations in the module. For example, `red s(p(p(0)))` returns `s(0)` for `Z`:

```
CafeOBJ> mod! Z{ ... }
-- defining module! Z....._..* done.
CafeOBJ> select Z
Z> red s(p(p(0))) .
-- reduce in Z : s(p(p(0)))
p(0) : Int
```

A special operation symbol `==` can be used for equational reasoning. The equality predicate `==` takes a pair of terms belonging to the same sort and returns `true` or `false`. When inputting `red` s `==` t, CafeOBJ reduces both terms into s' and t', and returns `true` if s' and t' are the same, and otherwise returns `false`:

```
Z> red s(p(0)) == p(s(0)) .
-- reduce in Z : s(p(0)) == p(s(0))
true : Bool
```

The equational reasoning with `red` (or `red _==_`) is sound, which means that if `red` t returns t' (or `red` t `==` t' returns `true`), then $SP \models t = t'$ holds. The equational reasoning is not complete for several reasons, for example, TRS may not be confluent, terms may include a variable, and a specification may not be tight. These reasons are discussed in Sections 3, 4, and 5.

2.3 Structured Specification

We briefly introduce the notion of specification imports. Details can be found in [8]. A specification can import a sub specification [1]. There are three import relations $\lhd_p$, $\lhd_e$, and $\lhd_u$, called protecting, extending, and using imports, respectively. $SP' \lhd_x SP$ means that SP imports SP' with mode x. We may use $\lhd$ as one of the three imports. Each imported specification is assumed to be declared with either the tight or loose denotation declaration. We present the properties of imports related to our research: (1) when SP imports SP' with protecting mode, any sort or operation symbol x in SP' is protected in SP, i.e. for any $M \in [SP]$, $M_x = M'_x$ for some $M' \in [SP']$, (2) the import relation is transitive, and the mode of the composed import is the weakest mode, where $\lhd_p$ is strongest and $\lhd_u$ is weakest, e.g., if $SP \lhd_p SP' \lhd_e SP''$, then $SP \lhd_e SP''$.

[1] SP' is a sub specification of SP, denoted by $SP' \subseteq SP$, iff $S_{SP'} \subseteq S_{SP}$, $\leq_{SP'} \subseteq \leq_{SP}$, $\Sigma_{SP'} \subseteq \Sigma_{SP}$ and $E_{SP'} \subseteq E_{SP}$.

In CafeOBJ, an import relation $SP' \lhd_p SP$ (or $SP' \lhd_e SP$, $SP' \lhd_u SP$) is described as $\mathtt{pr}(SP')$ (or $\mathtt{ex}(SP')$, $\mathtt{us}(SP')$). Note that import declarations should be irreflexive, e.g., CafeOBJ does not allow, for example, `mod! MOD{ pr(MOD)···}` and `mod! A{pr(B)···}` ··· `mod! B{pr(A)···}`. For a module with imports, SP_{MOD} includes, for example, the sorts, the operation symbols, and the equations described in the imported modules, as well as MOD itself. When declared only in MOD itself, the above are denoted as S_{MOD}, Σ_{MOD}, and E_{MOD}, respectively. Thus, when MOD imports $\overrightarrow{MOD_i}$ [2], $E_{SP_{MOD}} = E_{MOD} \cup \bigcup_i E_{SP_{MOD_i}}$, for example.

Example 3. The following is an example of specifications with imports:

```
mod! FUNN{ pr(Z) pr(FUN)
  op fn : Int Elt -> Elt
  var E : Elt   var X : Int
  eq fn(0, E) = E .
  eq fn(s(X), E) = f (fn(X, E)) .}
```

`FUNN` imports `Z` and `FUN` with the protecting mode. Thus, for example, $M_{\mathtt{Int}}$ is the set of integers (or its isomorphism) for any $M \in [SP_{\mathtt{FUNN}}]$. The operation symbol `fn` is interpreted into a function $M_{\mathtt{fn}}$ from $M_{\mathtt{Int}} \times M_{\mathtt{Elt}}$ to $M_{\mathtt{Elt}}$ satisfying $M_{\mathtt{fn}}(n, e) = f^n(e)$ if $n \geq 0$. Note that $M_{\mathtt{fn}}(n, e)$ can be any integer for $n < 0$.

3 Modular Equational Proof System

We next present an axiomatic semantics (Section 3) and an operational semantics (Section 4) for a modular algebraic specification language. The semantics presented in Sections 3 and 4 are similar to ordinary semantics, such as those presented in [8]. The difference is that we prepare notations to extract the part that corresponds to each submodule. The target language is the subset of CafeOBJ introduced in Section 2. The congruence relation $=_E$ for a set E of equations is defined as follows: $t =_E t'$ iff $t = t'$ can be derived from the reflexive, symmetric, transitive, congruent, and substitutive laws from E [8]. We redefine this congruence relation while maintaining the module structure. We define the congruence relation $=_{MOD}$ for each module MOD.

Definition 1. For a module MOD, the congruence relation $(\forall X)\_ =_{MOD} \_$ on $T(\Sigma_{SP_{MOD}}, X)$ (abbr. T) is defined as the smallest relation satisfying the following laws:

[**reflexivity**] [**symmetry**] [**transitivity**] For $s, t, u \in T$,

$$\frac{}{(\forall X)t =_{MOD} t} \quad \frac{(\forall X)s =_{MOD} t}{(\forall X)t =_{MOD} s} \quad \frac{(\forall X)s =_{MOD} u \;\; (\forall X)u =_{MOD} t}{(\forall X)s =_{MOD} t}$$

[**congruence**] For $f \in (\Sigma_{SP_{MOD}})_{s_1 \cdots s_n\, s}$ and $t_i, t'_i \in T_{s_i}$ $(i \in \{1, \ldots, n\})$,

$$\frac{(\forall X)t_1 =_{MOD} t'_1 \;\; \cdots \;\; (\forall X)t_n =_{MOD} t'_n}{(\forall X)f(t_1, \ldots, t_n) =_{MOD} f(t'_1, \ldots, t'_n)}$$

[2] We write $\overrightarrow{a_i}$ instead of $a_1, a_2, \ldots, a_n$.

[**substitutivity**] For $(\forall Y)t = t' \in E_{MOD}$ and $\theta : Y \to T$,

$$\frac{}{(\forall X)\theta(t) =_{MOD} \theta(t')}$$

[**import**] For modules MOD that import MOD' and $s, t \in T$, if $(\forall X)s =_{MOD'} t$, then

$$\frac{}{(\forall X)s =_{MOD} t}$$

The last two laws can be a leaf of a proof tree. [**substitutivity**] is an instance of an equation e that belongs to MOD itself. By [**import**], any equation derived from the submodule is also derivable.

Example 4. Figure 1 is a proof tree for $(\forall\emptyset)\mathtt{mod2}(\mathtt{s}(\mathtt{s}(\mathtt{p}(\mathtt{s}(\mathtt{0}))))) =_{\mathtt{FUNN}} \mathtt{0}$. We omit $(\forall X)$ for each equation, where $X_{\mathtt{Elt}} = \{\mathtt{E}\}$. Note that the left-most leaf comes from $\mathtt{s}(\mathtt{p}(\mathtt{s}(\mathtt{0}))) =_{\mathtt{Z}} \mathtt{s}(\mathtt{0})$.

$$\dfrac{\dfrac{\dfrac{}{\mathtt{s}(\mathtt{p}(\mathtt{s}(\mathtt{0}))) =_{\mathtt{FUNN}} \mathtt{s}(\mathtt{0})}}{\mathtt{fn}(\mathtt{s}(\mathtt{p}(\mathtt{s}(\mathtt{0}))),\mathtt{E}) =_{\mathtt{FUNN}} \mathtt{fn}(\mathtt{s}(\mathtt{0}),\mathtt{E})} \quad \dfrac{\dfrac{}{\mathtt{fn}(\mathtt{s}(\mathtt{0}),\mathtt{E}) =_{\mathtt{FUNN}} \mathtt{f}(\mathtt{fn}(\mathtt{0},\mathtt{E}))} \quad \dfrac{\dfrac{}{\mathtt{fn}(\mathtt{0},\mathtt{E}) =_{\mathtt{FUNN}} \mathtt{E}}}{\mathtt{f}(\mathtt{fn}(\mathtt{0},\mathtt{E})) =_{\mathtt{FUNN}} \mathtt{f}(\mathtt{E})}}{\mathtt{fn}(\mathtt{s}(\mathtt{0}),\mathtt{E}) =_{\mathtt{FUNN}} \mathtt{f}(\mathtt{E})}}{\mathtt{fn}(\mathtt{s}(\mathtt{p}(\mathtt{s}(\mathtt{0}))),\mathtt{E}) =_{\mathtt{FUNN}} \mathtt{f}(\mathtt{E})}$$

Fig. 1. A proof tree for $\mathtt{fn}(\mathtt{s}(\mathtt{p}(\mathtt{s}(\mathtt{0}))),\mathtt{E}) =_{\mathtt{FUNN}} \mathtt{f}(\mathtt{E})$

3.1 Soundness and Completeness of MEPS

We show the soundness of the modular equational proof system (abbr. MEPS), i.e., $s =_{MOD} t \Rightarrow SP_{MOD} \models s = t$, and gives a sufficient condition under which MEPS is complete, $s =_{MOD} t \Leftarrow SP_{MOD} \models s = t$. Let MOD be a module, and let E be the set of all equations in MOD and its imported modules, i.e., $E_{SP_{MOD}}$. It is trivial that $=_{MOD}$ and $=_E$ are exactly the same binary relation. Thus, the following properties hold [8].

Proposition 1. Let MOD be a module, $s, t \in T(\Sigma_{SP_{MOD}}, X)$. If $(\forall X)s =_{MOD} t$, then $SP_{MOD} \models s = t$.

Proposition 2. Let MOD be a tight and basic, i.e., with no explicit imports, module. Let $s, t \in T(\Sigma_{SP_{MOD}}, \emptyset)$. Then, $SP_{MOD} \models s = t \Leftrightarrow s =_{MOD} t$.

4 Modular Term Rewriting System

For a specification (Σ, E) and an equation $l = r \in E$, if l is not a variable and all variables in r occur in l, (Σ, E) (or just E) is called a TRS. In a TRS, equations are used as left-to-right rewrite rules. We may write $l \to r$ and R instead of $l = r$ and E when emphasizing rewrite rules. We propose an extension of TRSs for the module system, called a modular TRS (or MTRS), and an MTRS rewrite relation.

Definition 2. MTRSs are defined recursively as follows: an MTRS $\mathcal{R}$ is a pair $((\Sigma, R), A)$ of a TRS R and a set A of MTRSs satisfying the following: $\Sigma' \subseteq \Sigma$ for each MTRS $((\Sigma', R'), A') \in A$.

For a module MOD, the set E_{MOD} of equations described in MOD corresponds to the TRS of the first argument (Σ, R), and imported modules correspond to the second argument A. Since the rewrite rules (equations) in a module are constructed from the operation symbols declared in the module itself and *imported modules*, the condition $\Sigma' \subseteq \Sigma$ is needed. Basic modules correspond to MTRSs in which the second arguments are empty.

In order to assign an MTRS rewrite relation $\rightarrow_{\mathcal{R}}$, we introduce a positions set $O(t)$, a subterm $t|_p$ and a replacement term $t[u]_p$ as follows: $O(x) = \{\varepsilon\}$ and $O(f(t_1, \ldots, t_n)) = \{\varepsilon\} \cup \{i.p \in \mathcal{N}_+{}^* \mid p \in O(t_i)\}$. $t|_\varepsilon = t$ and $f(t_1, \ldots, t_n)|_{i.p} = t_i|_p$. $t[u]_\varepsilon = u$ and $f(t_1, \ldots, t_n)[u]_{i.p} = f(\ldots, t_{i-1}, t_i[u]_p, t_{i+1}, \ldots)$. The set of all maps from A to B is denoted by B^A. The reflexive transitive closure of $\rightarrow$ is denoted by $\rightarrow^*$.

Definition 3. Let $\mathcal{R} = (R, \{\mathcal{R}_i\}_{i=1,\ldots,n})$ be an MTRS. The MTRS rewrite relation $\rightarrow_{\mathcal{R}}$ is defined as follows:

$$s \rightarrow_{\mathcal{R}} t \overset{\text{def}}{\iff} \begin{cases} \exists(\forall X) l \rightarrow r \in R, \theta \in T^X, p \in O(s).(s|_p = \theta(l) \wedge t = s[\theta(r)]_p) \\ \text{or } \exists i \in \{1, \ldots, n\}.s \rightarrow_{\mathcal{R}_i} t. \end{cases}$$

The first part is the definition of the ordinary TRS rewrite relation $\rightarrow_R$ for a TRS R, and the latter part $(\exists i \in \{1, \ldots, n\}.s \rightarrow_{\mathcal{R}_i} t)$ corresponds to the rewrite relation of the imported modules.

Example 5. MTRSs $\mathcal{R}_{\mathtt{Z}} = ((\Sigma_{\mathtt{Z}}, E_{\mathtt{Z}}), \emptyset)$ and $\mathcal{R}_{\mathtt{FUN}} = ((\Sigma_{\mathtt{FUN}}, \emptyset), \emptyset)$ correspond to modules `Z` and `FUN`. MTRS $\mathcal{R}_{\mathtt{FUNN}} = ((\Sigma_{\mathtt{FUNN}} \cup \Sigma_{\mathtt{Z}} \cup \Sigma_{\mathtt{FUN}}, E_{\mathtt{FUNN}}), \{\mathcal{R}_{\mathtt{Z}}, \mathcal{R}_{\mathtt{FUN}}\})$ corresponds to the modules `FUNN`. $\mathtt{s(p(s(0))) \rightarrow_{\mathcal{R}_Z} s(0)}$ holds. Thus, $\mathtt{fn(s(p(s(0))),E)} \rightarrow_{\mathcal{R}_{\mathtt{FUNN}}} \mathtt{fn(s(0),E)} \rightarrow_{\mathcal{R}_{\mathtt{FUNN}}} \mathtt{f(fn(0,E))} \rightarrow_{\mathcal{R}_{\mathtt{FUNN}}} \mathtt{f(E)}$ holds.

Let MOD be a module importing $\overrightarrow{MOD_i}$. MTRS $\mathcal{R}_{MOD}$ is defined as the pair $(E_{MOD}, \overrightarrow{\mathcal{R}_{MOD_i}})$. We write $\mathcal{R}, \overrightarrow{\mathcal{R}_i}$ instead of $\mathcal{R}_{MOD}, \overrightarrow{\mathcal{R}_{MOD_i}}$ if no confusion exists. We hereinafter assume the existence of a corresponding module for each MTRS.

4.1 Soundness and Completeness of MTRS

When a (possibly infinite) sequence $\overrightarrow{s_i}$ of terms satisfies $s_i \rightarrow_{\mathcal{R}} s_{i+1}$ for each $i = 0, 1, 2, \ldots$, the sequence is referred to as a rewrite sequence, denoted by $s_0 \rightarrow_{\mathcal{R}} s_1 \rightarrow_{\mathcal{R}} s_2 \rightarrow_{\mathcal{R}} \cdots$. If $(s \rightarrow^* t$ and$jt \rightarrow u$ does not hold for any $u \in T$, t is called a $\rightarrow$-normal form (of s). We often omit $\rightarrow$. For a binary relation $\rightarrow$ written as an arrow, we define $\leftarrow = \{(a, b) \mid b \rightarrow a\}$ and $\leftrightarrow = \rightarrow \cup \leftarrow$. The reflexive and transitive closure of $\leftrightarrow_{\mathcal{R}}$ coincides with $=_{MOD}$. Terms a and b are joinable, and are denoted by $a \downarrow b$ when there exists c such that $a \rightarrow^* c$ and $b \rightarrow^* c$. $\rightarrow$ is confluent iff $b \downarrow c$ whenever $a \rightarrow^* b$, $a \rightarrow^* c$. $\rightarrow$ is terminating

iff there is no infinite rewrite sequence $s_0 \to s_1 \to \cdots$. $\to$ is convergent iff it is confluent and terminating. We define *equational reasoning by MTRS* as the following procedure: take terms s and t, reduce them into their $\to_{\mathcal{R}}$ normal forms, and return true if they are the same, and otherwise return false.

We show the soundness of the MTRS equational reasoning ($s \downarrow_{\mathcal{R}} t \Rightarrow s =_{MOD} t$) and the sufficient condition under which the MTRS equational reasoning is complete ($s \downarrow_{\mathcal{R}} t \Leftarrow s =_{MOD} t$). Let $\mathcal{R}$ be a MTRS, and let R the TRS union of all TRSs, including $\mathcal{R}$. It is trivial that the MTRS rewrite relation $\to_{\mathcal{R}}$ and the ordinary TRS rewrite relation $\to_R$ are exactly the same binary relation. Thus, the following properties hold [14,15].

Proposition 3. If $s \downarrow_{\mathcal{R}_{MOD}} t$, then $(\forall X) s =_{MOD} t$.

Proposition 4. Let MOD be a module such that $\to_{\mathcal{R}_{MOD}}$ is convergent. Let $s, t \in T(\Sigma_{SP_{MOD}}, X)$. Then, $(\forall X) s =_{MOD} t \Leftrightarrow s \downarrow_{\mathcal{R}_{MOD}} t$.

5 Equality Predicate

In CafeOBJ, we can use the equality predicate `_==_` not only in verification, but also in description as an operation symbol. t_1 `==` t_2 is a term for $t_1, t_2 \in T_s$. The equality predicate is included in a built-in module `BOOL`. The built-in module `BOOL` has a sort `Bool`, constants `true` and `false`, and operations such as `_and_`, `_or_`, and `_==_`. [3]. The rank of `_==_` is s s `-> Bool` for any sort s. It is not an ordinary operation symbol because it is polymorphic. In addition, the equality predicate `_==_` has another special quality in that it cannot be defined by equations or a TRS (even if the target sorts are fixed). Although we can give `eq X:`s `== X = true` for true cases, `eq X:`s `== Y:`s `= false` does not give false cases because t `==` t can be an instance of `X == Y`. The reduction command for a term including `_==_` is defined as follows: for a pattern s `==` t, reduce both terms, i.e., `red` s and `red` t, and replace the term with `true` if the results are same, and otherwise replace the term with `false`. The equality predicate denotes the equality of model values, i.e., $M_{==}(x, y) = (x = y)$ [8]. If we use the equality predicate as an operation symbol in the axiom, the equational reasoning is no longer sound because Propositions 2 and 4 do not hold without the assumptions. We next show examples in which $SP \models s = t$ does not hold, even if `red s == t` returns `true`.

Confluence: Consider SP with `eq a = b` and `eq a = c`. `red b == c` returns `false` because `b` and `c` are normal forms. Thus, `red (b == c) == false` returns `true`. However, $SP \models$ `b == c` $=$ `false` do not hold because `b = a = c`.

Denotation: For the loose module `FUN` (See Section 2.2), `red (f(E:Elt) == E) == false` returns `true` because `f(E)` and `E` are normal forms. However, $SP_{\texttt{FUN}} \not\models$

[3] We define a basic module (or specification) as a module without *explicit* imports. The *explicit* import means that the module has no import declaration, and does not use anything belonging to `BOOL`, i.e., the sort `Bool`, operations `true`, `false`, `_and_`, etc. Such modules can be considered as modules with no imports.

$(\forall\{\texttt{E}\})$ `(f(E) == E)` $=$ `false` because there exists $M \in [SP_{\texttt{FUN}}]$, which interprets `f` into the identity function, i.e., $M_{\texttt{f}}(x) = x$ for all $x \in M_{\texttt{Elt}}$.

Verification: Even if the specification is convergent as a TRS and is declared with a tight denotation, equational reasoning is still unsound for a specification with the equality predicate. Before showing a problematic example, we introduce a proof score, which is a basis for CafeOBJ verifications. Consider the following module: `mod! PROOF{ pr(Z) op n : -> Int }`. Since `Z` is protected, the constant `n` should be an integer. Then, the reduction command `red s(p(n)) == p(s(n))` in `PROOF` returns `true`, which means that $M \models \texttt{s}(\texttt{p}(\texttt{n})) = \texttt{p}(\texttt{s}(\texttt{n}))$ for any $M \in [SP_{\texttt{PROOF}}]$ from Proposition 1 and 3. For any integer $n \in Z_{\texttt{Int}}$, there exists $M \in [SP_{\texttt{PROOF}}]$ such that $M_{\texttt{n}} = n$. Thus, $Z_{\texttt{s}}(Z_{\texttt{p}}(n)) = Z_{\texttt{p}}(Z_{\texttt{s}}(n))$ holds for any $n \in Z_{\texttt{Int}}$. The theory of a proof using a constant as an arbitrary element, called *Theorem of Constants*, can be found in [11]. By the `open` command, we can declare a nameless module that imports the opened module with the protect mode. The following code, called a proof score, has the same meaning as the above proof:

```
CafeOBJ> open Z
-- opening module Z.. done.
%Z> op n : -> Int .
%Z> red s(p(n)) == p(s(n)) .
-- reduce in %Z : s(p(n)) == p(s(n))
true : Bool
```

Consider the following proof score for the module `ZERO` with the equality predicate (Section 1).

```
CafeOBJ> open ZERO
-- opening module ZERO.. done.
%ZERO> op n : -> Int .
%ZERO> red zero(n) == false .
-- reduce in %ZERO : zero(n) == false
true : Bool
```

The proof score means that `zero`(n) $=$ `false` for any integer n. However, this does not hold because 0 is an integer. The literature [11] mentions the problem of the equality predicate (Section 2.1.1). One solution given by [11] is to give a user-defined equality predicate for each sort needed. For example, the equality predicate `_is_` on `Nat`, which is defined as `0 : -> Nat` and `s : Nat -> Nat`, is defined by the four equations: `(0 is 0) = true`, `(s(N) is 0) = false`, `(0 is s(N)) = false`, and `(s(M) is s(N)) = M is N`, where `M` and `N` are variables [11]. However, it is not always possible for the user to find a suitable definition. For example, how should `_is_` be defined on `Int` for the specification `Z`? The equation `(s(N) is 0) = false` does not hold for `N` $=$ `p(0)` on `Int`. Moreover, the user should prove that each user-defined equality predicate actually denotes the equality on its target set.

5.1 Local Equality Predicate

To solve the problems of the equality predicate, we propose a new equality predicate, called the local equality predicate (LEP). The equality predicate implemented in CafeOBJ is hereinafter referred to as the global equality predicate (GEP). The LEP is defined for the specification language without the global equality predicate, which we introduced in Section 2, 3 and 4. For a module MOD, the specification SP_{MOD} is redefined as follows:

Definition 4. Assume that MOD imports $\overrightarrow{MOD_i}$. The specification SP_{MOD} is redefined by replacing the definition of $\Sigma_{SP_{MOD}}$ as follows: $\Sigma_{SP_{MOD}} = \Sigma_{MOD} \cup \bigcup_i \Sigma_{MOD_i} \cup \{\texttt{op \_=}MOD_i\texttt{=\_ : s s -> Bool} \mid \texttt{s} \in S_{MOD_i}\}$. $E_{SP_{MOD}} = E_{MOD} \cup \bigcup_i E_{MOD_i} \cup \{\texttt{eq (X:s =}MOD_i\texttt{= X) = true} \mid \texttt{s} \in S_{MOD_i}\}$. The other parts, such as $S_{SP_{MOD}}$, are not changed.

Denotation: For any SP_{MOD}-algebra M, LEP $\texttt{op \_=}MOD_i\texttt{=\_ : s s -> Bool}$ is interpreted in the equality on $M_{\texttt{s}}$.

MEPS: Add the following to Definition 1:

[LEP] For module MOD, which imports MOD' and $s, t \in T(\Sigma_{SP_{MOD}}, \emptyset)$, if $(\forall X)s \neq_{MOD'} t$, then

$$\overline{(\forall X)(s \texttt{ =}MOD'\texttt{= } t) =_{MOD} \texttt{false}}$$

MTRS: Add the following condition to Definition 2: MTRS $((\Sigma, R), A)$ satisfies the following: $\texttt{op \_=}MOD_i\texttt{=\_ : s s -> Bool} \in \Sigma$ for each $s \in S_{MOD_i}$ and $\mathcal{R}_i \in A$, and $\texttt{eq (X:s =}MOD_i\texttt{= X) = true} \in R$ for each $\mathcal{R}_i \in A$ for each $s \in S_{MOD_i}$ and $\mathcal{R}_i \in A$. Replace Definition 3 with the following:

$$s \to_{\mathcal{R}} t \overset{\text{def}}{\iff} \begin{cases} \exists (\forall X) l \to r \in R, \theta \in T^X, p \in O(s).(s|_p = \theta(l) \wedge t = s[\theta(r)]_p) \\ \text{or } \exists i \in \{1, \ldots, n\}.s \to_{\mathcal{R}_i} t \\ \text{or } \exists p \in O(s).(s|_p = (u \texttt{ =}MOD_i\texttt{= } v) \wedge t = s[\texttt{false}]_p \wedge u \not\downarrow_{\mathcal{R}_i} v) \end{cases}$$

Note that for both MEPS and MTRS, the `true` cases are given by simply adding an ordinary equation $\texttt{eq X =}MOD_i\texttt{= X = true}$.

5.2 Soundness of the Local Equality Predicate

Here, we present properties on the soundness of LEP. These properties can be proved from Proposition 2 and 4, and the proofs are omitted.

Theorem 1. [Soundness of MEPS with LEP] Let MOD be a module such that for any occurrence of $\texttt{=}MOD_i\texttt{=}$ in MOD, the module MOD_i is basic, tight, and imported with the protecting mode. Let $s, t \in T(\Sigma_{SP_{MOD}}, X)$. Then, $SP_{MOD} \models s = t$ if $(\forall X)s =_{MOD} t$.

Theorem 2. [Soundness of MTRS with LEP] Let MOD be a module such that for any occurrence of $\texttt{=}MOD_i\texttt{=}$ in MOD, the rewrite relation $\to_{\mathcal{R}_i}$ is convergent. Let $s, t \in T(\Sigma_{SP_{MOD}}, X)$. Then, $(\forall X)s =_{MOD} t$ if $s \downarrow_{\mathcal{R}} t$.

Corollary 1. Equational reasoning by MTRS for specifications with LEP is sound under the assumption of Theorems 1 and 2.

5.3 Sound Verification System

We next introduce a sound verification system (or rewrite engine) for specifications with LEP. Consider the assumption of Corollary 1 (i.e., Theorems 1 and 2). The tight denotation, the basic module, and the protecting import can be easily checked. Consider an MTRS $(R, \emptyset)$ corresponding to a basic module. $(R, \emptyset)$ can be regarded as an ordinary TRS R. For an ordinary TRS, many useful sufficient conditions and tools for termination have been proposed [14,15,10,1,4,6], and when assuming termination, the confluence property is decidable and can be proved using the critical pair method [14,15]. Thus, we can obtain a decidable procedure P to check the assumption of Corollary 1 by using termination provers. The reduction command for LEP is defined as follows. $s =_{MOD_i=} t$ is rewritten as follows: Reduce s and t into their normal forms s' and t', respectively. If s' and t' are same, then replace the equation with `true`. If s' and t' are not same, then check the conditions (1) $s', t' \in T(\Sigma_{SP_{\mathtt{MOD}_i}}, \emptyset)$ and (2) $P(MOD_i)$. If the conditions hold, then replace the equation with `false`, and otherwise return the equation as is. Then, from Corollary 1, the obtained reduction command is sound.

Example 6. We show the experiences of the reduction command for specifications with local equality predicates [4]. We modify `ZERO` as follows:

```
mod! ZERO{ pr(LEP-Z)
  op zero : Int -> Bool
  eq zero(X:Int) = (X =Z= 0) .}
```

where `LEP-Z` is the module having the local equality predicate `_=Z=_` on `Z` (Omit the definition). In the new `ZERO`, the global equality predicate `_==_` has been replaced with `_=Z=_`. For the terms `zero(s(p(0)))` and `zero(s(p(s(0))))`, the CafeOBJ system returns the correct answers `true` and `false`. We again try the proof score shown in Section 5 as follows:

```
CafeOBJ> open ZERO
-- opening module ZERO.. done.
%ZERO> op n : -> Int .
%ZERO> red zero(n) == false .
-- reduce in %ZERO : zero(n) == false
false : Bool
```

As a result of the LEP, an incorrect `true` is not returned for the above proof score. Note that `false` does not mean a disproof. The following is a proof score of `n` $\neq$ `0` $\Rightarrow$ `zero(n)` $=$ `false`, and returns `true`. Thus, it holds for any $M \in [SP_{\mathtt{M2}}]$.

```
ZERO> open ZERO
-- opening module ZERO.. done.
%ZERO> op n : -> Int .
%ZERO> eq n =Z= 0 = false .
```

[4] We have implemented each local equality predicate in CafeOBJ manually by using order sorts.

```
%ZERO> red zero(n) == false .
-- reduce in %ZERO : zero(n) == false
true : Bool
```

6 Applications

6.1 Application to Full CafeOBJ

The CafeOBJ specifications in the above sections are restricted in order to focus on the essential part of the problem of GEP. LEP can be applied to full CafeOBJ (or OBJ languages) straightforwardly, which include, for example, conditional equations, parameterized modules, behavioral specifications, and rewrite specifications.

6.2 Application of LEP

The assumption of Corollary 1 is not so restrictive. The tight denotation and the protecting import are necessary conditions. The convergence property is one of the properties that any algebraic specification is expected to satisfy. In particular, Maude, one of the OBJ languages, requires its functional modules (corresponding to CafeOBJ tight modules specifying a data type of a target system) to be convergent in order to obtain a sound rewriting engine for system specifications [5]. Since the import relation is transitive, the basic module is not so restrictive. For example, `_=Z=_` can be used in a module, which imports `FUNN` with the protecting mode, where `FUNN` imports `Z` with the protecting mode. The following `TREE` is an example outside the assumption:

```
mod! TREE{ pr(Z) [Int < Tree]
  op __ : Tree Tree -> Tree  }
```

This is a specification of trees having leaves that are integers: Term `s(0)` is a tree from `[Int < Tree]`. Term `(p(0) s(0)) p(0)` is another example of trees. `TREE` is not basic, and `Tree` is defined in `TREE`. The LEP on trees is outside the assumption of Corollary 1. However, if necessary, we can describe the corresponding basic specification satisfying the assumption as follows:

```
mod! TREE{ [Zero < Int < Tree]
  op 0 : -> Zero
  op s p : Int -> Int
  op __ : Tree Tree -> Tree
  eq s(p(X:Int)) = X .
  eq p(s(X:Int)) = X .}
```

6.3 Applications of MEPS and MTRS

MEPS and MTRS are useful not only for dealing with the equality predicates, but also for the introduction of several functions to modular specification languages. For example, built-in modules are treated well by our framework rather than the ordinary framework. Some built-in modules have only signatures and

do not have equations (axioms), thus, the ordinary TRS does not treat them directly. The meaning of operation symbols are implemented in other low-level languages, e.g., Common Lisp for CafeOBJ built-in modules. For example, a built-in module `NAT` has constants `0`, `1`, `2`, ..., operation symbols, such as `_+_`, `_*_`, and the expression x `+` y, for example, is reduced to the result of the evaluation of the Common Lisp expression `(+` x y`)`. In our modular framework, we simply define $=_{\texttt{NAT}}$ and $\rightarrow_{\texttt{NAT}}$ from the implementation of `NAT` in order to obtain the axiomatic and operational semantics of specifications with the built-in module `NAT`. Similarly, our framework can be used while implementing a specification. When we have implemented submodules of a given large specification, we may obtain a specification combined with those implementations and can perform some execution tests for the ongoing implementation, or verifications for the semi-implemented specification. Integrating other verification techniques, such as model-checking, with a rewriting-based verification is another possible use of our framework.

7 Conclusion

We proposed the modular equational proof system and the modular term rewriting system, which are suitable for algebraic specification languages with a module system. We also proposed the local equality predicate and showed its soundness (Corollary 1 and Section 5.3). The problem of the global equality predicate is well-known in the CafeOBJ community, and it has not been used in the recent practical specifications. The current CafeOBJ system also supports another equality predicate `=`, which is implemented by simply `eq (X = X) = true`, where `X` is a variable. Many case studies have been succeeded with the above simple equality predicate. However, because it does not support false cases, we have to manually provide false cases needed for verification. The local equality predicate solves the problem of the equality predicate while maintaining its advantages.

For the TRS area, MTRS with LEP is related to the conditional TRS (CTRS) with negative conditions. A conditional rewrite rule in CTRS is written as $l \rightarrow r$ if $\bigwedge l_i = r_i$. When $l_i \downarrow_R r_i$, an instance of l is replaced with the instance of r (there are several definitions of CTRS rewrite relations. See [14,15]). If a negative equation is included in the condition part, it is not easy to prove the soundness of the CTRS. A conditional rewrite rule can be described in CafeOBJ as a conditional equation in which the condition is a term of the sort `Bool`. We can give a `Bool` condition term as $(u_1 = \texttt{M1} = \texttt{v}_1)$ `and` $\cdots$ `and` $(u_n = \texttt{Mn} = \texttt{v}_\texttt{n})$ for positive equations and $\texttt{not}(\texttt{u}'_1 = \texttt{M1}' = \texttt{v}'_1)$ `and` $\cdots$ `and` $\texttt{not}(\texttt{u}'_\texttt{n} = \texttt{Mn}' = \texttt{v}'_\texttt{n})$ for negative equations. We can say that MTRS with the local equality predicate gives one solution of the difficulty of dealing with negative conditions in CTRS.

References

1. AProVE, `http://www-i2.informatik.rwth-aachen.de/AProVE/`
2. BOBJ, `http://www.cs.ucsd.edu/groups/tatami/bobj/`

3. CafeOBJ, http://www.ldl.jaist.ac.jp/cafeobj/
4. CiME, http://cime.lri.fr/
5. Maude, http://maude.cs.uiuc.edu/
6. Tyrolean Termination Tool, http://cl2-informatik.uibk.ac.at/ttt/
7. Burstall, R.M., Goguen, J.A.: The Semantics of CLEAR, A Specification Language. In: Bjorner, D. (ed.) Abstract Software Specifications. LNCS, vol. 86, pp. 292–332. Springer, Heidelberg (1980)
8. Diaconescu, R., Futatsugi, K.: CafeOBJ Report. World Scientific, Singapore (1998)
9. Futatsugi, K., Goguen, J.A., Jouannaud, J.-P., Meseguer, J.: Principles of OBJ2. In: POPL. Proceedings of the 12th ACM Symposium on Principles of Programming Languages, pp. 52–66. ACM Press, New York (1985)
10. Giesl, J., Thiemann, R., Schneider-Kamp, P., Falke, S.: Automated Termination Proofs with AProVE. In: van Oostrom, V. (ed.) RTA 2004. LNCS, vol. 3091, pp. 210–220. Springer, Heidelberg (2004)
11. Goguen, J.A., Malcolm, G.: Algebraic Semantics of Imperative Programs, Massachusetts Institute of Technology (1996)
12. Goguen, J.A., Winkler, T., Meseguer, J., Futatsugi, K., Jouannaud, J.-P.: Software Engineering with OBJ: Algebraic Specification in Action. Kluwers Academic Publishers, Boston, MA (2000) (chapter Introducing OBJ*)
13. Meinke, K., Tucker, J.V.: Universal Algebra. In: Abramsky, S., Gabbay, D.M., Maibaum, T.S.E. (eds.) Handbook of Logic in Computer Science: Background - Mathematical Structures, vol. 1, pp. 189–411. Clarendon Press, Oxford (1992)
14. Ohlebusch, E.: Advanced topics in term rewriting. Springer, Heidelberg (2002)
15. Terese: Term Rewriting Systems. Cambridge Tracts in Theoretical Computer Science, vol. 55. Cambridge University Press, Cambridge (2003)
16. Wirsing, M.: Algebraic Specification. In: Handbook of Theoretical Computer Science. Formal Models and Sematics (B), pp. 675–788 (1990)