Introducing CafeOBJ specifications in the Full Maude database. Integrating CafeOBJ with the Constructor-based Inductive Theorem Prover—Programmer Guide

Adrián Riesco

Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid

August 2013
(Revised September 26, 2013)

Abstract

CafeOBJ and Maude are sister languages of the OBJ language, and two of the most advanced formal specification languages for systems verification. Although both of them have a similar syntax and semantics, different usages and applications have been developed for them. Hence, a tool combining both languages would be very useful for exploiting the specific features of each of them.

We present here the details of the implementation to load CafeOBJ specifications into the Full Maude. We first show how to define the syntax, the functions for parsing, and the rules to deal with general commands. Then, we present how to extend a specific tool, the Constructor-based Inductive Theorem Prover (CITP), to deal with CafeOBJ specifications. This is achieved by creating an attribute that distinguishes the current language, combining the syntax of both tools, and defining rules that deal with the commands when our new language is chosen.

Contents

1 Introducing CafeOBJ modules into the Full Maude database

We present in this section the main modules required for dealing with CafeOBJ specifications [2]. First, we will see how to define the syntax, and how to parse the terms to obtain the Maude modules [1]. Then, we will show how to pretty print these terms and how to create new commands and rules and will be added in addition to the ones already defined in Full Maude.

1.1 Syntax

The module `CafeBUBBLES` defines the sorts `@CafeBubble@` for bubbles (that is, terms that can take any form, like the lefthand side of an equation), `@CafeBubble@` for tokens (like sorts), and `@NeCafeTokenList@` for nonempty lists of tokens:

```
fmod CafeBUBBLES is
  including QID-LIST .

  sorts @CafeBubble@ @CafeToken@ @NeCafeTokenList@ .

  op CafeBubble : QidList -> @CafeBubble@ [special (id-hook Bubble (1 -1 ( ))
                           op-hook qidListSymbol (__ : QidList QidList ~> QidList)
                           op-hook qidSymbol (<Qids> : ~> Qid))] .

  op CafeToken : Qid -> @CafeToken@ [special (id-hook Bubble (1 1)
                           op-hook qidSymbol (<Qids> : ~> Qid)
                           id-hook Exclude(. [] < { } ( )))] .

  op neCafeTokenList : QidList -> @NeCafeTokenList@ [special (id-hook Bubble (1 -1)
                           op-hook qidListSymbol (__ : QidList QidList ~> QidList)
                           op-hook qidSymbol (<Qids> : ~> Qid)
                           id-hook Exclude(. { } ))] .
endfm
```

The module `Cafe-ATTRIBUTES` defines the possible attributes that can be used in operators and equations:

```
fmod Cafe-ATTRIBUTES is
  pr CafeBUBBLES .
```

It defines the sorts `@CafeAttr@` for a single attribute and `@CafeAttrList@` for lists of attributes:

```
  sorts @CafeAttr@ @CafeAttrList@ .
  subsorts @CafeAttr@ < @CafeAttrList@ .
```

The attributes are defined following the CafeOBJ syntax. The unary attributes are all defined in the same way, the identity attributes require take a bubble as argument, while the precedence attribute takes a token:

```
  op __ : @CafeAttrList@ @CafeAttrList@ -> @CafeAttrList@ [ctor assoc] .
  ops assoc associative l-assoc r-assoc comm commutative constr nonexec
      idem idempotent : -> @CafeAttr@ [ctor] .
  op id:`(_`) : @CafeBubble@ -> @CafeAttr@ [ctor] .
  op idr:`(_`) : @CafeBubble@ -> @CafeAttr@ [ctor] .
  op prec:_ : @CafeToken@ -> @CafeAttr@ [ctor] .
```

Although the `metadata` attribute is not currently available for CafeOBJ specifications, it might be useful, so we support it by defining the appropriate operator. However, it can also be defined in a comment, as explained in [3]:

```
  op metadata_ : @CafeToken@ -> @CafeAttr@ [ctor] .
endfm
```

The syntax must also include the commands that we want to use for CafeOBJ specifications. These commands are defined in the `TRANSLATION-COMMANDS` module, which imports the `COMMANDS` module from Full Maude. To add new commands the user must define them here and then specify their behavior in the module `CAFE2MAUDE-DATABASE-HANDLING` described in Section 1.4:

```
fmod TRANSLATION-COMMANDS is
  inc COMMANDS .
```

We have defined three commands:

- The first one will force the translation to be done without using the modifications presented in [3], that drop some requirements from the importations modes and on the usage of theories to allow a wider range of CafeOBJ specifications to be translated:

  ```
  op strict`translation`on`. : -> @Command@ .
  ```

- Analogously, the second one will allow these modifications:

  ```
  op strict`translation`off`. : -> @Command@ .
  ```

- The third one will require a CafeOBJ module to be shown:

  ```
    op original`CafeOBJ`module_. : @Token@ -> @Command@ .
  endfm
  ```

The module `CafeMETA-SIGN` defines the syntax for CafeOBJ modules:

```
fmod CafeMETA-SIGN is
  including FULL-MAUDE-SIGN .
  including Cafe-ATTRIBUTES .
```

It first defines all the required sorts, and the subsort relations between them:

```
  sorts @CafeMODULE@ @HiddenSortDecl@ @VisibleSortDecl@ @CafeOpDecl@
        @CafeImportDecl@ @CafeType@ @CafeTypeList@ @CafeSortList@ @CafeSort@
        @BehaviorEquationDecl@ @CafeDeclList@ @CafeEqDecl@ @CafeVarDecl@
        @CafeSubSortRel@ @CafeLDeclList@ @CafeModExp@ @CafeParameter@ @CafeParameters@
        @CafeInterface@ @CafeViewDecl@ @CafeViewDeclList@ @CafeTransDecl@ @CafeViewId@
        @CafeViewIdList@ @ReductionDecl@ .

  subsort @CafeToken@ < @CafeSort@ < @CafeType@ .
  ...
```

Then it defines the syntax of every possible construction in CafeOBJ. For example, we can define the syntax for:

- Hidden sorts, which receive a token and create a `@HiddenSortDecl@`:

  ```
  op *`[_`]* : @CafeToken@ -> @HiddenSortDecl@ [ctor] .
  ```

- View identifiers, which can be either:

– On the fly view declarations, receiving a module expression and a declaration list:

```
op view`to_`{_`} : @CafeModExp@ @CafeViewDeclList@ -> @CafeViewId@ [ctor] .
```

– The abbreviated version of the previous declaration, which does not require the `view to` keywords:

```
op _`{_`} : @CafeModExp@ @CafeViewDeclList@
            -> @CafeViewId@ [ctor prec 15] .
```

– A view identifier assigned to a specific module expression:

```
op _<=_ : @CafeModExp@ @CafeViewId@
          -> @CafeViewId@ [ctor prec 20] .
```

– Finally, lists of view identifiers are created by using the operator `_,_`. Note that this operator is the one with the lower precedence, since it must not interfer with the previous declarations:

```
op _,_ : @CafeViewIdList@ @CafeViewIdList@
         -> @CafeViewIdList@ [ctor assoc prec 25] .
```

- Importations, including all the possible variants:

```
op protecting`(_`) : @CafeModExp@ -> @CafeImportDecl@ [ctor] .
op pr`(_`) : @CafeModExp@ -> @CafeImportDecl@ [ctor] .
op extending`(_`) : @CafeModExp@ -> @CafeImportDecl@ [ctor] .
op ex`(_`) : @CafeModExp@ -> @CafeImportDecl@ [ctor] .
op including`(_`) : @CafeModExp@ -> @CafeImportDecl@ [ctor] .
op inc`(_`) : @CafeModExp@ -> @CafeImportDecl@ [ctor] .
op using`(_`) : @CafeModExp@ -> @CafeImportDecl@ [ctor] .
op us`(_`) : @CafeModExp@ -> @CafeImportDecl@ [ctor] .
```

- Sort and subsort declarations:

```
op `[_`] : @CafeSortList@ -> @VisibleSortDecl@ [ctor prec 5] .
op `[_`] : @CafeSubSortRel@ -> @VisibleSortDecl@ [ctor prec 5] .
```

- Subsort relations:

```
op _<_ : @CafeSortList@ @CafeSortList@ -> @CafeSubSortRel@ [ctor] .
op _<_ : @CafeSortList@ @CafeSubSortRel@ -> @CafeSubSortRel@ [ctor] .
```

- Equations. Note that the label or the possible attributes are not included into the operator definition. Instead, we will deal with the bubble defining the lefthand side to check whether there is a label. Similarly, we will analyze the last bubble looking for attributes, such as `nonexec`:

```
op eq_=_. : @CafeBubble@ @CafeBubble@ -> @CafeEqDecl@ [ctor] .
op ceq_=_if_. : @CafeBubble@ @CafeBubble@ @CafeBubble@ -> @CafeEqDecl@ [ctor] .
op cq_=_if_. : @CafeBubble@ @CafeBubble@ @CafeBubble@ -> @CafeEqDecl@ [ctor] .
```

- Transitions. Analogously to the case above, we do not declare explicitly the label or the attributes of the transitions:

```
  op trans_=>_. : @CafeBubble@ @CafeBubble@ -> @CafeTransDecl@ [ctor] .
  op trns_=>_. : @CafeBubble@ @CafeBubble@ -> @CafeTransDecl@ [ctor] .
  op ctrans_=>_if_. : @CafeBubble@ @CafeBubble@ @CafeBubble@
                    -> @CafeTransDecl@ [ctor] .
  op ctrns_=>_if_. : @CafeBubble@ @CafeBubble@ @CafeBubble@
                   -> @CafeTransDecl@ [ctor] .
  op btrans_=>_. : @CafeBubble@ @CafeBubble@ -> @CafeTransDecl@ [ctor] .
  op btrns_=>_. : @CafeBubble@ @CafeBubble@ -> @CafeTransDecl@ [ctor] .
  op bctrans_=>_if_. : @CafeBubble@ @CafeBubble@ @CafeBubble@
                     -> @CafeTransDecl@ [ctor] .
  op bctrns_=>_if_. : @CafeBubble@ @CafeBubble@ @CafeBubble@
                    -> @CafeTransDecl@ [ctor] .
```

- Predicates. In this case the sorts for the definition are more specific than in the cases above (where we just used bubbles), so we distinguish whether attributes are declared or not:

```
  op pred_:_`{_`}. : @CafeToken@ @CafeTypeList@ @CafeAttrList@
                   -> @CafeOpDecl@ [ctor] .
  op pred_:_. : @CafeToken@ @CafeTypeList@ -> @CafeOpDecl@ [ctor] .
```

- Module and view declarations. Note that they do not have the exact syntax used by CafeOBJ. This distinction is obtained after a pre-processing stage that makes sure that there is no clash with the Maude syntax for modules and views:

```
  op cmod*_`{_`} : @CafeInterface@ @CafeDeclList@ -> @CafeMODULE@ [ctor] .
  op cmod!_`{_`} : @CafeInterface@ @CafeLDeclList@ -> @CafeMODULE@ [ctor] .
  op cview_from_to_`{_`} : @CafeToken@ @CafeModExp@ @CafeModExp@
                         @CafeViewDeclList@ -> @CafeMODULE@ [ctor] .
```

- Open-close environment. This block is composed by a module expression and a list of declarations, possibly including reduction commands. Note that we have introduced an extra dot to ease the parsing; this dot will be added during the preprocessing stage, so the user is not required to type it:

```
  op copen_._close : @CafeModExp@ @CafeLDeclList@ -> @CafeMODULE@ [ctor] .
endfm
```

The module META-CAFE2MAUDE-SIGNATURE contains the metapresented signature required by CafeOBJ, which extends the one for Full Maude:

```
fmod META-CAFE2MAUDE-SIGNATURE is
 including META-FULL-MAUDE-SIGN .

 op CafeGRAMMAR : -> FModule .
 eq CafeGRAMMAR = addImports((including 'CafeMETA-SIGN .)
                            (including 'TRANSLATION-COMMANDS .), GRAMMAR ) .
endfm
```

The modules CafeSIGN and OPERATOR-Cafe, just describe auxiliary functions for dealing with CafeOBJ modules, like functions for adding new sorts, equations, or transitions, or for obtaining these values.

1.2 Parsing

The module `CafeDECL-PARSING` is in charge of parsing a term built following the syntactical constructions presented above:

```
fmod CafeDECL-PARSING is
  inc UNIT-DECL-PARSING .
  inc OPERATOR-Cafe .
  pr MAP{Qid, Qid} * (sort Map{Qid, Qid} to SortMap) .
```

It defines the sort `CafeParseResult` to return the result of the parsing process. The is composed of:

- A term of sort `ParseDeclResult`. This sort, defined in Full Maude, keeps the module obtained thus far, another module still containing bubbles, and a set of operator declarations standing for the declaration of variables on the fly.
- A list of quoted identifiers, that will propagate the errors found during the parsing process.
- A database, that will be updated with the new module or view if the parsing is successful.

```
  sort CafeParseResult .
  op <_,_,_> : ParseDeclResult QidList Database -> CafeParseResult [ctor] .
```

The function `parseCafeRen` translates CafeOBJ renamings. Note that the renaming for hidden sorts is translated as a renaming for standard sorts, since there is no hidden sorts in Maude:

```
  op parseCafeRen : Term -> Term .
  ceq parseCafeRen('__[T, T']) = '_`,_[T'', T3]
   if T'' := parseCafeRen(T) /\
      T3 := parseCafeRen(T') .
  ceq parseCafeRen('sort_->_.[T, T']) = 'sort_to_[T'', T3]
   if T'' := sort2sort(T) /\
      T3 := sort2sort(T') .
  ceq parseCafeRen('hsort_->_.[T, T']) = 'sort_to_[T'', T3]
   if T'' := sort2sort(T) /\
      T3 := sort2sort(T') .
  eq parseCafeRen('op_->_.['CafeBubble[T], 'CafeBubble[T']]) =
                                    'op_to_['token[T], 'token[T']] .
  eq parseCafeRen('bop_->_.['CafeBubble[T], 'CafeBubble[T']]) =
                                    'op_to_['token[T], 'token[T']] .
  eq parseCafeRen(T) = T [owise] .
```

The function `sort2sort` translates CafeOBJ tokens into Maude tokens for sorts, while `parseCafeViewExp` translates view tokens:

```
  op sort2sort : Term -> Term .
  eq sort2sort('CafeToken[T]) = 'sortToken[T] .
  eq sort2sort(T) = T [owise] .

  op parseCafeViewExp : Term -> ViewExp .
  eq parseCafeViewExp('token[T]) = 'viewToken[T] .
  eq parseCafeViewExp(T) = T [owise] .
```

The function `parseCafeDecl` uses the function `parseCafeModExp` to parse de module expression. Once this expression is obtained it uses the Full Maude function

```
op parseCafeDecl : Term Module Module OpDeclSet Database -> CafeParseResult .
ceq parseCafeDecl('protecting`(_`)[T], PU, U, VDS, DB) = < PDR, nil, DB' >
 if < T', DB' > := parseCafeModExp(T, DB) /\
    PDR := parseDecl('protecting_.[T'], PU, U, VDS) .
ceq parseCafeDecl('pr`(_`)[T], PU, U, VDS, DB) = < PDR, nil, DB' >
 if < T', DB' > := parseCafeModExp(T, DB) /\
    PDR := parseDecl('protecting_.[T'], PU, U, VDS) .
...
```

The function parseCafeModExp returns a term of sort ParseResult. This sort contains a database and, depending the context where it is applied, a module expression, a view expression, or a list of terms:

```
sort ParseResult .
op <_,_> : ModuleExpression Database -> ParseResult [ctor] .
op <_,_> : ViewExp Database -> ParseResult [ctor] .
op <_,_> : TermList Database -> ParseResult [ctor] .
```

Parsing simple module expressions, summations, and expressions with renamings is straightforward:

```
op parseCafeModExp : Term Database -> ParseResult .
eq parseCafeModExp('CafeToken[T], DB) = < 'token[T], DB > .
eq parseCafeModExp('token[T], DB) = < 'token[T], DB > .
ceq parseCafeModExp('_+_[T, T'], DB) = < '_+_[T'', T'''], DB'' >
 if < T'', DB' > := parseCafeModExp(T, DB) /\
    < T''', DB'' > := parseCafeModExp(T', DB') .
ceq parseCafeModExp('_*`{_`}[T, T'], DB) = < '_*`(_`)[T'', T'''], DB' >
 if < T'', DB' > := parseCafeModExp(T, DB) /\
    T''' := parseCafeRen(T') .
```

Parsing a module expression involving view expression requires the database, because we need the list of parameters to deal with on-the-fly view declarations:

```
ceq parseCafeModExp('_`(_`)[T, T'], DB) = < '_`{_`}[T'', T'''], DB'' >
 if < T'', DB' > := parseCafeModExp(T, DB) /\
    M := getTopModule(parseModExp(T''), DB') /\
    PDL := getPDL(M) /\
    < T''', DB'' > := parseCafeViewExp(PDL, sortViewId(PDL, T'), DB') .
```

Parsing a view identifier requires the parameter list of the module being instantiated, the view identifier itself, and the database.

```
op parseCafeViewExp : ParameterDeclList Term Database -> ParseResult .
eq parseCafeViewExp(PDL, 'token[T], DB) = < 'viewToken[T], DB > .
eq parseCafeViewExp(PDL, '_<=_[T, T'], DB) = parseCafeViewExp(PDL, T', DB) .
ceq parseCafeViewExp(PDL, '_`,_[TL], DB) = < '_`,_[TL'], DB' >
 if < TL', DB' > := parseCafeViewExp*(PDL, TL, DB) .
```

When parsing views declared on the fly we create a new auxiliary view, add it to the database, and return the new database and the name of the auxiliary view:

```
ceq parseCafeViewExp(Q :: ME, 'view`to_`{_`}[T, T'], DB) =
                                                < 'viewToken[upTerm(Q')], DB' >
 if Q' := getNewName(DB, 0) /\
    T'' := 'token[upTerm(Q')] /\
    TV := 'view_from_to_is_endv[T'', 'token[upTerm(ME)], T, maps2maps(T')] /\
    DB' := procView(TV, DB) .
```

```
ceq parseCafeViewExp(Q :: ME, '_`{_`}[T, T'], DB) = < 'viewToken[upTerm(Q')], DB' >
 if Q' := getNewName(DB, 0) /\
    T'' := 'token[upTerm(Q')] /\
    TV := 'view_from_to_is_endv[T'', 'token[upTerm(ME)], T, maps2maps(T')] /\
    DB' := procView(TV, DB) .
```

The auxiliary functions used thus far are defined as follows:

- The function `parseCafeViewExp*` traverses the terms in the list, pairing them with the parameters from the module being instantiated:

```
op parseCafeViewExp* : ParameterDeclList TermList Database -> ParseResult .
eq parseCafeViewExp*(PDL, empty, DB) = < empty, DB > .
ceq parseCafeViewExp*((Q :: ME, PDL), (T, TL), DB) = < (T', TL'), DB'' >
 if < T', DB' > := parseCafeViewExp(Q :: ME, T, DB) /\
    < TL', DB'' > := parseCafeViewExp(PDL, TL, DB') .
```

- The function `maps2maps` transforms CafeOBJ mappings into Maude mappings. This is achieved by first computing the mappings for sorts and variables, removing the variable declarations, which are not allowed in Maude, and then applying an auxiliary `maps2maps` function with 3 arguments:

```
op maps2maps : Term -> Term .
ceq maps2maps(T) = maps2maps(T', SM, VM)
 if SM := getSortMap(T) /\
    VM := getVarMap(T) /\
    T' := removeVarDecls(T) .
```

- The function `getSortMap` traverses the mappings, transforming them into Maude mappings:

```
op getSortMap : Term -> SortMap .
ceq getSortMap('__[T, T']) = SM, SM'
 if SM := getSortMap(T) /\
    SM' := getSortMap(T') .
eq getSortMap('sort_->_.[T, T']) = getSort(T) |-> getSort(T') .
eq getSortMap('hsort_->_.[T, T']) = getSort(T) |-> getSort(T') .
eq getSortMap(T) = empty [owise] .
```

  where `getSort` just extracts the quoted identifier from the term:

```
op getSort : Term ~> Term .
eq getSort('CafeToken[T]) = downQid(T) .
```

- Similarly, `getVarMap` traverses the term creating a mapping between variables and their sort. Note that we use `vvar` and `vvars` instead of `var` and `vars`. This is due to a pre-processing step that aims to distinguish between these variables and the ones in modules:

```
op getVarMap : Term -> SortMap .
ceq getVarMap('__[T, T']) = VM, VM'
 if VM := getVarMap(T) /\
    VM' := getVarMap(T') .
ceq getVarMap('vvar_:_.['neCafeTokenList[T], 'CafeToken[T']]) =
                                                       createMap*(QIL, Q)
```

```
 if QIL := downQidList(T) /\
    Q := downQid(T') .
ceq getVarMap('vvars_:_.['neCafeTokenList[T], 'CafeToken[T']]) =
                                                     createMap*(QIL, Q)
 if QIL := downQidList(T) /\
    Q := downQid(T') .
eq getVarMap(T) = empty [owise] .
```

where `createMap*` just maps all the variables in the first argument to the sort given as second argument:

```
op createMap* : QidList Qid -> SortMap .
eq createMap*(nil, Q) = empty .
eq createMap*(Q QIL, Q') = Q |-> Q', createMap*(QIL, Q') .
```

- The function `maps2maps` with 3 arguments translates CafeOBJ mappings into Maude mappings. Sort renamings only require changing the syntax:

```
op maps2maps : Term SortMap SortMap -> Term .
ceq maps2maps('__[T, T'], SM, VM) = '__[T'', T3]
 if T'' := maps2maps(T, SM, VM) /\
    T3 := maps2maps(T', SM, VM) .
ceq maps2maps('sort_->_.[T, T'], SM, VM) = 'sort_to_.[T'', T3]
 if T'' := sort2sort(T) /\
    T3 := sort2sort(T') .
ceq maps2maps('hsort_->_.[T, T'], SM, VM) = 'sort_to_.[T'', T3]
 if T'' := sort2sort(T) /\
    T3 := sort2sort(T') .
```

However, mapping operators might require a more complex translation, because they can include mappings to terms. We check whether the lefthand side contains variables. If it does not contain, then it is translated as an operator mapping; otherwise, it is mapped to a term:

```
ceq maps2maps('op_->_.['CafeBubble[T], 'CafeBubble[T']], SM, VM) =
                      if T == T''
                      then 'op_to_.['token[T], 'token[T3]]
                      else 'op_to`term_.['bubble[T''], 'bubble[T3]]
                      fi
 if T'' := updateTermLHS(T, SM, VM) /\
    T3 := updateTermRHS(T', SM, VM) .
```

Finally, behavioral operators are transformed into standard operators:

```
ceq maps2maps('bop_->_.['CafeBubble[T], 'CafeBubble[T']], SM, VM) =
                                'op_to_.['bubble[T''], 'bubble[T3]]
 if T'' := updateTermLHS(T, SM, VM) /\
    T3 := updateTermRHS(T', SM, VM) .
eq maps2maps(T, SM, VM) = T [owise] .
```

- The function `updateTermLHS` traverses the constants in the term and, if we find a variable previously defined, its sort is attached:

```
op updateTermLHS : Term SortMap SortMap -> Term .
eq updateTermLHS(Q[TL], SM, VM) = Q[updateTermLHS*(TL, SM, VM)] .
eq updateTermLHS(V, SM, VM) = V .
ceq updateTermLHS(C, SM, VM) = upTerm(Q')
 if Q := downQid(C) /\
    VM[Q] =/= undefined /\
    Q' := qid(string(Q) + ":" + string(VM[Q])) .
eq updateTermLHS(T, SM, VM) = T [owise] .
```

- The function updateTermRHS also attaches the sort of the variables, but it takes into account that *the names of the sorts might have changed due to the mappings*. Hence, it looks for the sort name in the sort mapping and replaces it if required:

```
op updateTermRHS : Term SortMap SortMap -> Term .
eq updateTermRHS(Q[TL], SM, VM) = Q[updateTermRHS*(TL, SM, VM)] .
eq updateTermRHS(V, SM, VM) = V .
ceq updateTermRHS(C, SM, VM) = upTerm(Q')
 if Q := downQid(C) /\
    VM[Q] =/= undefined /\
    SM[VM[Q]] == undefined /\
    Q' := qid(string(Q) + ":" + string(VM[Q])) .
ceq updateTermRHS(C, SM, VM) = upTerm(Q')
 if Q := downQid(C) /\
    VM[Q] =/= undefined /\
    SM[VM[Q]] =/= undefined /\
    Q' := qid(string(Q) + ":" + string(SM[VM[Q]])) .
eq updateTermRHS(T, SM, VM) = T [owise] .
```

- The functions updateTermLHS* and updateTermRHS* just traverse the list, applying the appropriate function to each element:

```
ops updateTermLHS* : TermList SortMap SortMap -> TermList .
eq updateTermLHS*(empty, SM, VM) = empty .
eq updateTermLHS*((T, TL), SM, VM) = updateTermLHS(T, SM, VM),
                                     updateTermLHS*(TL, SM, VM) .

op updateTermRHS* : TermList SortMap SortMap -> TermList .
eq updateTermRHS*(empty, SM, VM) = empty .
eq updateTermRHS*((T, TL), SM, VM) = updateTermRHS(T, SM, VM),
                                     updateTermRHS*(TL, SM, VM) .
```

- The function removeVarDecls removes the variable declarations from the term, by first removing them and then creating a new term with the rest of the declarations:

```
op removeVarDecls : Term -> Term .
eq removeVarDecls(T) = buildNotVarDecl(getNotVarDecl(T)) .
```

- The function getNotVarDecl checks that the operator at the top is not a variable declaration:

```
op getNotVarDecl : Term -> TermList .
ceq getNotVarDecl('__[Q[TL], Q'[TL']]) = T, T'
 if Q =/= 'vvar_:_. /\
    Q =/= 'vvars_:_. /\
    Q' =/= 'vvar_:_. /\
```

```
    Q' =/= 'vvars_:_. /\
    T := getNotVarDecl(Q[TL]) /\
    T' := getNotVarDecl(Q'[TL']) .
ceq getNotVarDecl('__[Q[TL], Q'[TL']]) = T
 if Q == 'vvar_:_. or-else Q == 'vvars_:_. /\
    Q' =/= 'vvarrs_:_. /\
    Q' =/= 'vvars_:_. /\
    T := getNotVarDecl(Q'[TL']) .
ceq getNotVarDecl('__[Q[TL], Q'[TL']]) = T
 if Q =/= 'vvarrs_:_. /\
    Q =/= 'vvars_:_. /\
    Q' == 'vvar_:_. or-else Q' == 'vvars_:_. /\
    T := getNotVarDecl(Q[TL]) .
eq getNotVarDecl(T) = T [owise] .
```

- The function `buildNotVarDecl` distinguishes whether the argument is a singleton list or not, in order to use the `__` operator:

```
op buildNotVarDecl : TermList ~> Term .
eq buildNotVarDecl(T) = T .
ceq buildNotVarDecl((T, TL)) = '__[T, buildNotVarDecl(TL)]
 if TL =/= empty .
```

- The function `sortViewId` is in charge of sorting the parameters, so they do not rely on the naming features of CafeOBJ. If only one parameter is used or the term does not use syntactic sugar, then it is kept the same way. Otherwise an alternative function is used:

```
op sortViewId : ParameterDeclList Term -> Term .
ceq sortViewId(PDL, Q[T, TL]) = Q[T, TL]
 if Q =/= '_`,_ or-else not usesSugar(T) .
eq sortViewId(PDL, Q[TL]) = sortViewIdAux(PDL, TL) .
```

where `sortViewIdAux` looks for the appropriate view identifiers by traversing the list of parameters:

```
op sortViewIdAux : ParameterDeclList TermList -> TermList .
eq sortViewIdAux(nil, TL) = empty .
ceq sortViewIdAux((Q :: ME, PDL), TL) = find(Q, TL), sortViewIdAux(PDL, TL)
 if PDL == nil .
eq sortViewIdAux((Q :: ME, PDL), TL) = '_`,_[find(Q, TL),
                                       sortViewIdAux(PDL, TL)] [owise] .
```

- The function `getPDL` extracts the parameter list from a module:

```
op getPDL : Module ~> ParameterDeclList .
eq getPDL(fmod Q{PDL} is IL sorts SS . SSDS OPDS MAS EqS endfm) = PDL .
eq getPDL(mod Q{PDL} is IL sorts SS . SSDS OPDS MAS EqS RIS endm) = PDL .
```

- `getNewName` checks in the database whether there already exists a view with name created by `createViewName`. If true, then we try with the next natural number, otherwise this name is used:

```
op getNewName : Database Nat -> Qid .
ceq getNewName(DB, N) = if getView(Q, DB) :: View
                        then getNewName(DB, s(N))
                        else Q
                        fi
 if Q := createViewName(N) .
```

where `createViewName` just creates a new name starting by `OTF-VIEW` [3], which stands for "on the fly view", and followed by a natural number:

```
op createViewName : Nat -> Qid .
eq createViewName(N) = qid("OTF-VIEW" + string(N, 10)) .
```

- The function `find` looks for the given quoted identifier, standing for a parameter, inside a list of terms:

```
op find : Qid TermList -> Term .
eq find(Q, ('_`,_[T, T'], TL)) = find(Q, (T, T', TL)) .
eq find(Q, ('_<=_['token[T], T'], TL)) = if Q == downQid(T)
                                         then T'
                                         else find(Q, TL)
                                         fi .
```

- Finally, `usesSugar` checks whether the notation `_<=_`, used to state the name of the parameter corresponding to the view identifier, is being used:

```
op usesSugar : Term -> Bool .
eq usesSugar('_<=_[TL]) = true .
eq usesSugar(T) = false [owise] .
```

The parsing process continues by parsing sorts. Hidden sorts are translated as standard Maude sorts:

```
ceq parseCafeDecl('*`[_`]*['CafeToken[T]], PU, U, VDS, DB) = < PDR, nil, DB >
 if PDR := parseDecl('sort_.['sortToken[T]], PU, U, VDS) .
```

We distinguish the operator at the top when dealing with sort declarations.

- When only one sort is declared it is parsed and added to the temporal modules with the Full Maude function `parseDecl`:

```
ceq parseCafeDecl('`[_`]['CafeToken[T]], PU, U, VDS, DB) = < PDR, nil, DB >
 if T' := addSortToken('CafeToken[T]) /\
    PDR := parseDecl('sorts_.[T'], PU, U, VDS) .
```

- When we have a list of sorts without subsort declaration they are just parsed and added to the current module:

```
ceq parseCafeDecl('`[_`]['__[T, T']], PU, U, VDS, DB) = < PDR, nil, DB >
 if T'' := addSortToken(T) /\
    T''' := addSortToken(T') /\
    PDR := parseDecl('sorts_.['__[T'', T''']], PU, U, VDS) .
```

- Finally, when we find a subsort relation both terms, the sorts are add to the sort and the subsort relation (which might be multiple) is added to the module thus obtained:

```
ceq parseCafeDecl('`[_`]['_<_[T, T']], PU, U, VDS, DB) = < PDR, nil, DB >
 if T'' := addSortToken(T) /\
    T''' := addSortToken(T') /\
    TS := sub2sort('_<_[T, T']) /\
    < PU' ; U' ;  VDS' > := parseDecl('sorts_.[TS], PU, U, VDS) /\
    PDR := parseDecl('subsorts_.['_<_[T'', T''']], PU', U', VDS') .
```

The auxiliary functions used for parsing sorts are:

- `addSortToken`, which transforms CafeOBJ tokens into Maude tokens for sorts:

```
op addSortToken : Term -> Term .
eq addSortToken('__[T, T']) = ('__[addSortToken(T), addSortToken(T')]) .
eq addSortToken('_<_[T, T']) = ('_<_[addSortToken(T), addSortToken(T')]) .
eq addSortToken('CafeToken[T]) = ('sortToken[T]) .
```

- `sub2sort`, which flattens a subsort relation to add all the sorts to the current module:

```
op sub2sort : Term -> Term .
eq sub2sort('_<_[T, T']) = combine2sort(sub2sort(T), sub2sort(T')) .
eq sub2sort('__['CafeToken[T], T']) = ('__['sortToken[T], sub2sort(T')]) .
eq sub2sort('CafeToken[T]) = ('sortToken[T]) .
```

- `combine2sort`, which puts together two terms:

```
op combine2sort : Term Term -> Term .
eq combine2sort('sortToken[T], T') = '__['sortToken[T], T'] .
eq combine2sort('__[T, T''], T''') = combine2sort(T'', '__[T, T''']) .
```

We show now how to parse operator declarations. For the declaration of a single operator with attributes we transform the list of sorts in the arity with `addSortToken`, and then add the operator declaration with `parseDecl`:

```
ceq parseCafeDecl('op_:_->_`{_`}.['CafeToken[T], T', 'CafeToken[T''], T'''],  PU, U,
                  VDS, DB) = < PDR, nil, DB >
 if T4 := addSortToken(T') /\
    PDR := parseDecl('op_:_->_`[_`].['token[T], T4, 'sortToken[T''],
                     map2MaudeAttr(T''')], PU, U, VDS) .
```

The declaration of constants simplifies the task, since the arity does not appear:

```
ceq parseCafeDecl('op_:`->_`{_`}.['CafeToken[T], 'CafeToken[T''], T'''],  PU, U, VDS,
                  DB) = < PDR, nil, DB >
 if PDR := parseDecl('op_:`->_`[_`].['token[T], cafeType2maudeType(T''),
                                    map2MaudeAttr(T''')], PU, U, VDS) .
```

Multiple operators with the same arity and coarity are just transformed into a nonempty list of Maude tokens, and then introduced into the current module:

```
ceq parseCafeDecl('ops_:_->_.['neCafeTokenList[T], T', 'CafeToken[T'']], PU, U, VDS,
                  DB) = < PDR, nil, DB >
 if T4 := addSortToken(T') /\
    PDR := parseDecl('ops_:_->_.['neTokenList[T], T4, cafeType2maudeType(T'')],
                     PU, U, VDS) .
```

Predicates are parsed in a similar way, since they are transformed into operators with coarity Bool, which must be meta-represented as a token:

```
ceq parseCafeDecl('pred_:_`{_`}.['CafeToken[T], T', T''''],  PU, U, VDS, DB) =
                                                              < PDR, nil, DB >
 if T4 := addSortToken(T') /\
    PDR := parseDecl('op_:_->_`[_`].['token[T], T4, 'sortToken[''Bool.Qid],
                 map2MaudeAttr(T''')], PU, U, VDS) .
```

The auxiliary functions used here are:

- The function tokenList2token transforms operator defintions of the form _ _, which are not allowed by Maude, into the equivalent __ operator:

```
op tokenList2token : TermList -> Qid .
op tokenList2token : TermList String -> Qid .

eq tokenList2token(TL) = tokenList2token(TL, "") .
eq tokenList2token(('CafeToken[T], TL), St) =
                     tokenList2token(TL, St + string(downQid(T))) .
eq tokenList2token(empty, St) = qid(St) .
```

- The function map2MaudeAttr translates a list of terms written using CafeOBJ syntax into the same list using Maude syntax. The first equation deals with the yuxtaposition operator at the top:

```
op map2MaudeAttr : TermList -> TermList .
eq map2MaudeAttr('__[TL]) = '__[map2MaudeAttr(TL)] .
```

  While the rest of equations just translates the attribute and continue with the rest of the list, until the empty list is reached:

```
eq map2MaudeAttr(('constr.@CafeAttr@, TL)) = 'ctor.@Attr@, map2MaudeAttr(TL) .
eq map2MaudeAttr(('associative.@CafeAttr@, TL)) = 'assoc.@Attr@,
                                                  map2MaudeAttr(TL) .
eq map2MaudeAttr(('assoc.@CafeAttr@, TL)) = 'assoc.@Attr@, map2MaudeAttr(TL) .
...

eq map2MaudeAttr(empty) = empty .
```

- The function cafeType2maudeType transforms CafeOBJ error types into Maude kinds:

```
op cafeType2maudeType : Term -> Term .
eq cafeType2maudeType('__[TL]) = '__[cafeTypes2maudeTypes(TL)] .
ceq cafeType2maudeType(T) = '`[_`]['sortToken[upTerm(Q')]]
 if Q := downQid(T) /\
    St := string(Q) /\
    0 == find(St, "?", 0) /\
    St' := substr(St, 1, length(St)) /\
    Q' := qid(St') .
eq cafeType2maudeType(T) = 'sortToken[T] [owise] .
```

  where the function cafeTypes2maudeTypes just traverses the list of terms, applying the function cafeType2maudeType to each term:

```
op cafeTypes2maudeTypes : TermList -> TermList .
eq cafeTypes2maudeTypes(empty) = empty .
eq cafeTypes2maudeTypes((T, TL)) = cafeType2maudeType(T),
                                   cafeTypes2maudeTypes(TL) .
```

Variables declared with the keyword `var` are just translated as `vars` declarations.

```
ceq parseCafeDecl('var_:_.['neCafeTokenList[T], T'], PU, U, VDS, DB) =
                                                    < PDR, QIL, DB' >
 if < PDR, QIL, DB' > := parseCafeDecl('vars_:_.['neCafeTokenList[T], T'],
                                  PU, U, VDS, DB) .
```

Otherwise, variables are parsed by adding them, with their sort, to the set of operators used to parse terms in the current module:

```
ceq parseCafeDecl('vars_:_.['neCafeTokenList[T], 'CafeToken[T']], PU, U, VDS, DB) =
                                                               < PDR, nil, DB >
 if PDR := < PU ; U ; VDS parseVars(downQidList(T), parseType('sortToken[T'])) > .
```

Equations and transitions are parsed in a similar way, so we do not show all the variations. An unconditional equation is parsed by adding the on-the-fly variables in the righthand side to the set of current variables, and transforming the attributes and then using `parseDecl`:

```
ceq parseCafeDecl('eq_=_.['CafeBubble[T], 'CafeBubble[T']], PU, U, VDS, DB) =
                                      < PDR, nil, DB >
 if QIL := downQidList(T) /\
    VDS' := VDS opDeclSetFromQidList(QIL) /\
    T'' := cafeEqAtS2maudeEqAts(T') /\
    PDR := parseDecl('eq_=_.['bubble[T], 'bubble[T'']], PU, U, VDS') .
```

Conditional equations are parsed in the same way, since the condition is dealt inside the `parseDecl` function:

```
ceq parseCafeDecl('ceq_=_if_.['CafeBubble[T], 'CafeBubble[T'], 'CafeBubble[T'']],
                  PU, U, VDS, DB) = < PDR, nil, DB >
 if QIL := downQidList(T) /\
    VDS' := VDS opDeclSetFromQidList(QIL) /\
    T''' := cafeEqAtS2maudeEqAts(T'') /\
    PDR := parseDecl('ceq_=_if_.['bubble[T], 'bubble[T'], 'bubble[T''']],
                     PU, U, VDS') .
```

Transitions follow the same approach, although in this case the statement parsed by `parseDecl` is a Maude rule:

```
ceq parseCafeDecl('trans_=>_.['CafeBubble[T], 'CafeBubble[T']], PU, U,
                  VDS, DB) = < PDR, nil, DB >
 if QIL := downQidList(T) /\
    VDS' := VDS opDeclSetFromQidList(QIL) /\
    T'' := cafeEqAtS2maudeEqAts(T') /\
    PDR := parseDecl('rl_=>_.['bubble[T], 'bubble[T'']], PU, U, VDS') .
```

The auxiliary functions for parsing equations and transitions are:

- `cafeEqAtS2maudeEqAts`, which just applies `cafeEqAtS2maudeEqAts*` if it finds any attribute:

```
op cafeEqAtS2maudeEqAts : Term -> Term .
eq cafeEqAtS2maudeEqAts('__[TL]) = '__[cafeEqAtS2maudeEqAts*(TL)] .
eq cafeEqAtS2maudeEqAts(T) = T [owise] .
```

where cafeEqAtS2maudeEqAts traverses the list and transforms the possible attributes appearing in equations and transitions:

```
op cafeEqAtS2maudeEqAts* : TermList -> TermList .
eq cafeEqAtS2maudeEqAts*((TL, '''{.Qid, ''nonexec.Qid, TL'', '''}.Qid)) =
                    TL, '''[.Qid, ''nonexec.Qid, TL'', ''']. Qid .
eq cafeEqAtS2maudeEqAts*((TL, '''{.Qid, ''metadata.Qid, TL'', '''}.Qid)) =
                    TL, '''[.Qid, ''metadata.Qid, TL'', ''']. Qid .
eq cafeEqAtS2maudeEqAts*(TL) = TL [owise] .
```

- opDeclSetFromQidList, which extracts an OpDeclSet from a list of quoted identifiers to extend the variable set with the variables defined on the fly in CafeOBJ:

```
op opDeclSetFromQidList : QidList -> OpDeclSet .
eq opDeclSetFromQidList(nil) = none .
ceq opDeclSetFromQidList(Q QL) = op qid(St') : nil -> qid(St'') [none] .
                                 opDeclSetFromQidList(QL)
 if Q =/= ': /\
    St := string(Q) /\
    N := find(St, ":", 0) /\
    St' := substr(St, 0, N) /\
    St'' := substr(St, s(N), length(St)) .
eq opDeclSetFromQidList(Q QL) = opDeclSetFromQidList(QL) [owise] .
```

```
endfm
```

The module CafePARSER is in charge of parsing complete modules and views:

```
mod CafePARSER is
  inc CafeDECL-PARSING .
  pr DATABASE-HANDLING .
```

It builds a new term of sort CafeParseResult returning the updated database and a list of quoted identifiers reporting the errors found during the parsing process:

```
  op <_,_> : Database QidList -> CafeParseResult [ctor] .
```

The constants errModName and paramThWarn will report specific errors:

```
  op errModName : -> QidList .
  eq errModName = '\n '\r 'ERROR: '\o 'The 'module 'name 'is 'not 'a
                 'valid 'identifier. .

  op paramThWarn : -> QidList .
  eq paramThWarn = '\n '\r 'Warning: '\o 'Parameterized 'theories 'are 'not 'allowed
                  'in 'Maude. '\n 'The 'module 'was 'introduced 'with '\g 'tight '\o
                  'semantics. '\n .
```

The function in charge of parsing modules is procCafeMod. It receives as arguments the term to be parsed and the current Full Maude database. It just duplicates the term to be parsed and calls to procCafeMod2:

```
op procCafeMod : Term Database -> CafeParseResult .
eq procCafeMod(T, DB) = procCafeMod2(T, T, DB) .
```

The function procCafeMod2 distinguishes between modules with loose and tight semantics. Modules with tight semantics, will be translated as Maude modules, and hence we just use the function procCafeMod3, propagating the parameters if required. Note that we use an empty module, emptyFModule, indicating that it has tight semantics:

```
op procCafeMod2 : Term Term Database -> CafeParseResult .
eq procCafeMod2(T, 'cmod!_`{_`}['CafeToken[T'], T''], DB) =
                    procCafeMod3(T, 'CafeToken[T'], empty, T'',
                                 emptyFModule, DB) .
eq procCafeMod2(T, 'cmod!_`{_`}['_`(_`)[T', T''], T3], DB) =
                    procCafeMod3(T, T', T'', T3, emptyFModule, DB) .
```

When parsing modules with loose semantics first introduce them as a theory but, in order to accept a wider range of CafeOBJ modules, we also introduce it into the database as a module, adding the suffix -MODCAFE to its name. Note that we do not allow parameterized theories, so we use the paramThWarn message when they are used. Also note that we use an empty module or thery depending of the semantics we want to use:

```
ceq procCafeMod2(T, 'cmod*_`{_`}['CafeToken[T'], T''], DB) =
      if QIL == nil
      then procCafeMod3(T, T3, empty, T'', emptyFModule, DB')
      else < DB', QIL >
      fi
 if < DB', QIL > := procCafeMod3(T, 'CafeToken[T'], empty, T'',
                                 emptyFTheory, DB) /\
    QI := downQid(T') /\
    QI' := qid(string(QI) + "-MODCAFE") /\
    T3 := 'CafeToken[upTerm(QI')] .
ceq procCafeMod2(T, 'cmod*_`{_`}['_`(_`)['CafeToken[T'], T''], T3], DB) =
      if QIL == nil
      then < DB', paramThWarn >
      else < DB', QIL >
      fi
 if < DB', QIL > := procCafeMod3(T, 'CafeToken[T'], T'', T3, emptyFModule, DB) .
```

In other case an error is returned:

```
eq procCafeMod2(T, Q[T', T''], DB) = < DB, errModName > [owise] .
```

The function procCafeMod3 deals with parameterized modules. If the current modules is not parameterized we just set the name of the module and apply the procCafeMod4 function:

```
op procCafeMod3 : Term Term Term Term Module Database -> CafeParseResult .
ceq procCafeMod3(T, 'CafeToken[T'], empty, T'', U, DB) =
       procCafeMod4(T, T'', setName(U, QI), setName(U, QI), none, DB)
 if QI := downQid(T') .
```

When the module is parameterized we set the name of the module and then parse the parameters to introduce them into the current module. We use the parseParList function from Full Maude, which returns a parameter list from a term.

```
ceq procCafeMod3(T, 'CafeToken[T'], PL, T'', U, DB) =
       procCafeMod4(T, T3, setPars(setName(U, QI), parseParList(PL')),
                    setName(U, QI), none, DB)
```

```
if PL =/= empty /\
   PL' := cafeParam2maudeParam(PL) /\
   QI := downQid(T') /\
   QIL := cafeParamNames(PL) /\
   T3 := paramSortsMap(T'', QIL) .
```

In other case an error is returned:

```
eq procCafeMod3(T, T', PL, T'', U, DB) = < DB, errModName > [owise] .
```

The auxiliary functions required to deal with parameters are:

- `cafeParam2maudeParam`, which translates the parameter declaration to Maude syntax:

```
op cafeParam2maudeParam : Term -> Term .
eq cafeParam2maudeParam('_::_['CafeToken[T], T']) = '_::_['token[T], T'] .
ceq cafeParam2maudeParam('_`,_[T, T']) = '_`,_[T'', T3]
 if T'' := cafeParam2maudeParam(T) /\
    T3 := cafeParam2maudeParam(T') .
```

- `cafeParamNames`, which extracts the name of the parameters:

```
op cafeParamNames : Term -> QidList .
eq cafeParamNames('_::_['CafeToken[T], T']) = downQid(T) .
ceq cafeParamNames('_`,_[T, T']) = QIL QIL'
 if QIL := cafeParamNames(T) /\
    QIL' := cafeParamNames(T') .
```

- `paramSortsMap`, which transforms the qualified sorts in CafeOBJ syntax into qualified sorts in Maude syntax. It renames constants (which includes the metarepresentation of variables) by using the names of the parameters obtained with `cafeParamNames`. If the function is a composed term, it applies `paramSortsMap*`:

```
op paramSortsMap : Term QidList -> Term .
eq paramSortsMap(Q[TL], QIL) = Q[paramSortsMap*(TL, QIL)] .
```

Variables are not modified:

```
eq paramSortsMap(V, QIL) = V .
```

For constants, we distinguish whether they stand for a constant (including sorts) or for a variable. When they stand for a constant, (i.e., the String `":"` cannot be found) we split the term looking for the `"."` String, which is used in CafeOBJ to qualify sorts, and built it again by using the `"$"` used by Maude:

```
ceq paramSortsMap(C, QIL) = upTerm(QI)
 if Q := downQid(C) /\
    St := string(Q) /\
    find(St, ":", 0) == notFound /\
    N := find(St, ".", 0) /\
    St' := substr(St, 0, N) /\
    St'' := substr(St, N + 1, length(St)) /\
    Q' := qid(St'') /\
    Q' in QIL /\
    QI := qid(St'' + "$" + St') .
```

When the constant stands for a variable, we proceed in a similar way but taking into account that the name of the variable must be placed first in both cases:

```
ceq paramSortsMap(C, QIL) = upTerm(QI)
 if Q := downQid(C) /\
    St := string(Q) /\
    N := find(St, ":", 0) /\
    N' := find(St, ".", 0) /\
    St' := substr(St, 0, N + 1) /\
    St'' := substr(St, N + 1, _-_(N', N + 1)) /\
    St''' := substr(St, N' + 1, length(St)) /\
    Q' := qid(St''') /\
    Q' in QIL /\
    QI := qid(St' + St''' + "$" + St'') .
```

In other case, the constant is not modified:

```
eq paramSortsMap(C, QIL) = C [owise] .
```

- `paramSortsMap*`, which just traverses the list, applying `paramSortsMap` to each element:

```
op paramSortsMap* : TermList QidList -> TermList .
eq paramSortsMap*(empty, QIL) = empty .
eq paramSortsMap*((T, TL), QIL) = paramSortsMap(T, QIL), paramSortsMap*(TL, QIL) .
```

- `_in_`, which looks for a quoted identifier in a list:

```
op _in_ : Qid QidList -> Bool .
eq Q in nil = false .
eq Q in Q QIL = true .
eq Q in QIL = false [owise] .
```

The function `procCafeMod4` traverses the module, applying the function `parseCafeDecl` shown above to each statement:

```
op procCafeMod4 : Term Term Module Module OpDeclSet Database
                  -> CafeParseResult .
ceq procCafeMod4(T, '__[T', T''], PU, U, VDS, DB) =
      if QIL == nil
      then procCafeMod4(T, T'', preModule(PDR), unit(PDR), vars(PDR), DB')
      else < DB,  QIL >
      fi
 if < PDR, QIL, DB' > := parseCafeDecl(T', PU, U, VDS, DB) .
```

When only one statement remains it is parsed and the module is evaluated by means of the `evalPreModule` function:

```
ceq procCafeMod4(T, F[TL], PU, U, VDS, DB) =
     if QIL == nil
     then < evalPreModule(preModule(PDR), unit(PDR), vars(PDR),
                          insTermModule(getName(U), T, DB')), nil >
     else < DB, QIL >
     fi
 if F =/= '__ /\
    < PDR, QIL, DB' > := parseCafeDecl(F[TL], PU, U, VDS, DB) .
```

The function procCafeView is in charge of processing views. It just translates the view and introduces it into the database:

```
  op procCafeView : Term Database -> CafeParseResult .
  ceq procCafeView(T, DB) = < DB', nil >
   if T' := view2view(T) /\
      DB' := procView(T', DB) .
```

where the auxiliary functions behave as follows:

- view2view translates the term to Maude syntax, and then applies the maps2maps function shown above to the body:

```
  op view2view : Term -> Term .
  eq view2view('cview_from_to_`{_`}[T, T', T'', T3]) =
        'view_from_to_is_endv[token2token(T), T', T'', maps2maps(T3)] .
```

- token2token translates a CafeOBJ token into a Maude token:

```
  op token2token : Term -> Term .
  eq token2token('CafeToken[T]) = 'token[T] .
  eq token2token(T) = T [owise] .
```

```
endm
```

1.3 Pretty printing

Once the modules are parsed, it might be interesting to print them. However, we cannot print them from the corresponding Maude module, since we have lost information about things like hidden sorts, behavioral equations, etc. For this reason, we will show how to print the term standing for the original CafeOBJ specification. The module CAFE-PRETTY-PRINT is in charge of printing:

```
mod CAFE-PRETTY-PRINT is
  pr CafePARSER .
```

We fix the Maude options for printing in the printOpts constant:

```
  op printOpts : -> PrintOptionSet .
  eq printOpts = mixfix number rat format .
```

The character preceding or following a scape character is usually printed without separation, which worsens the legibility. To prevent the system from doing it, we use the function addSpace, which adds extra space if required:

```
  op addSpace : QidList -> QidList .
  eq addSpace(QIL) = addSpaceL(addSpaceR(QIL)) .
```

where the auxiliary functions addSpaceL and addSpaceR add a space at the left and the right of the list, respectively:

```
op addSpaceL : QidList -> QidList .
eq addSpaceL('`( QIL) = ' '`( QIL .
eq addSpaceL('`[ QIL) = ' '`[ QIL .
eq addSpaceL('`{ QIL) = ' '`{ QIL .
eq addSpaceL(QIL) = QIL [owise] .

op addSpaceR : QidList -> QidList .
eq addSpaceR(QIL '`)) = QIL '`) ' .
eq addSpaceR(QIL '`]) = QIL '`] ' .
eq addSpaceR(QIL '`}) = QIL '`} ' .
eq addSpaceR(QIL) = QIL [owise] .
```

The function `printCond` prints a condition. It traverses each specific condition until `nil` is reached. We just show the equality case, where both terms are printed by using the `printOpts` constant above:

```
op printCond : Module Condition -> QidList .
eq printCond(M, nil) = nil .
eq printCond(M, T = T' /\ C) = metaPrettyPrint(M, T, printOpts) '=
                               metaPrettyPrint(M, T', printOpts)
                               if C == nil
                               then nil
                               else '/\ printCond(M, C)
                               fi .
 ...
```

The function `printCafeModule` is in charge of printing CafeOBJ modules. We only distinguish cases to print the appropriate keyword, but the rest of the methods are common for both kinds of modules:

```
op printCafeModule : Term Module Database Bool -> QidList .
ceq printCafeModule('cmod!_`{_`}[T, T'], M, DB, B) =
                 '\n 'mod! printCafeName(DB, M, T) '`{
                 '\n first(printCafeBody*(paramSortsMap(T', PL), M, DB, none, PL))
                 '\n '`}
 if PL := paramNames(T) .
ceq printCafeModule('cmod*_`{_`}[T, T'], M, DB, B) =
                 '\n 'mod* printCafeName(DB, M, T) '`{
                 '\n first(printCafeBody*(paramSortsMap(T', PL), M, DB, none, PL))
                 '\n '`}
 if PL := paramNames(T) .
```

where the function `paramNames` just extracts the parameter names from the term:

```
op paramNames : Term -> QidList .
eq paramNames('_`(_`)[T, T']) = cafeParamNames(T') .
```

The function `printCafeName` is in charge of printing the header of the module. It distinguishes between all the possible module expressions:

```
op printCafeName : Database Module Term -> QidList .
eq printCafeName(DB, M, 'CafeToken[T]) = downQid(T) .
eq printCafeName(DB, M, 'token[T]) = downQid(T) .
ceq printCafeName(DB, M, '_`(_`)[T, T']) = QIL '`( QIL' '`) '
 if QIL := printCafeName(DB, M, T) /\
    QIL' := printCafeViewExp(DB, M, T') .
ceq printCafeName(DB, M, '_*`{_`}[T, T']) = QIL '* ' '`{ QIL' '`}
 if QIL := printCafeName(DB, M, T) /\
```

```
    QIL' := printCafeRen(T') .
ceq printCafeName(DB, M, '_+_[T, T']) = QIL '+ QIL'
 if QIL := printCafeName(DB, M, T) /\
    QIL' := printCafeName(DB, M, T') .
```

The following auxiliary functions are required by `printCafeName`:

- `printCafeViewExp`, which prints any view expression:

```
op printCafeViewExp : Database Module Term -> QidList .
eq printCafeViewExp(DB, M, 'token[T]) = downQid(T) .
ceq printCafeViewExp(DB, M, '_`,_[T, T']) = QIL '`, ' QIL'
 if QIL := printCafeViewExp(DB, M, T) /\
    QIL' := printCafeViewExp(DB, M, T') .
ceq printCafeViewExp(DB, M, '_::_[T, T']) = QIL ':: QIL'
 if QIL := printCafeName(DB, M, T) /\
    QIL' := printCafeName(DB, M, T') .
ceq printCafeViewExp(DB, M, '_<=_[T, T']) = QIL '<= QIL'
 if QIL := printCafeName(DB, M, T) /\
    QIL' := printCafeViewExp(DB, M, T') .
ceq printCafeViewExp(DB, M, 'view`to_`{_`}[T, T']) =
                                   'view 'to QIL ' '`{ ' QIL' ' '`}
 if QIL := printCafeName(DB, M, T) /\
    QIL' := printCafeMaps(DB, M, T') .
ceq printCafeViewExp(DB, M, '_`{_`}[T, T']) = QIL ' '`{ ' QIL' ''`}
 if QIL := printCafeName(DB, M, T) /\
    QIL' := printCafeMaps(DB, M, T') .
```

- `printCafeMaps`, which prints the possible mappings appearing in views:

```
op printCafeMaps : Database Module Term -> QidList .
ceq printCafeMaps(DB, M, '__[T, T']) = QIL QIL'
 if QIL := printCafeMaps(DB, M, T) /\
    QIL' := printCafeMaps(DB, M, T') .
ceq printCafeMaps(DB, M, 'sort_->_.[T, T']) = 'sort Q 'to Q'
 if Q := printCafeName(DB, M, T) /\
    Q' := printCafeName(DB, M, T') .
ceq printCafeMaps(DB, M, 'hsort_->_.[T, T']) = 'hsort Q 'to Q'
 if Q := printCafeName(DB, M, T) /\
    Q' := printCafeName(DB, M, T') .
ceq printCafeMaps(DB, M, 'op_->_.['CafeBubble[T], 'CafeBubble[T']]) =
                                                     'op QIL '-> QIL'
 if QIL := metaPrettyPrint(M, T, printOpts) /\
    QIL' := metaPrettyPrint(M, T', printOpts) .
```

- `printCafeRen`, which is in charge of printing renamings:

```
op printCafeRen : Term -> QidList .
ceq printCafeRen('__[T, T']) = QIL '`, ' QIL'
 if QIL := printCafeRen(T) /\
    QIL' := printCafeRen(T') .
ceq printCafeRen('sort_->_.[T, T']) = 'sort QIL '-> QIL'
 if QIL := printCafeSort(T) /\
    QIL' := printCafeSort(T') .
ceq printCafeRen('hsort_->_.[T, T']) = 'hsort QIL '-> QIL'
 if QIL := printCafeSort(T) /\
    QIL' := printCafeSort(T') .
```

```
ceq printCafeRen('op_->_.[T, T']) = 'op QIL '-> QIL'
 if QIL := printCafeTerm(T) /\
    QIL' := printCafeTerm(T') .
ceq printCafeRen('bop_->_.[T, T']) = 'op QIL '-> QIL'
 if QIL := printCafeTerm(T) /\
    QIL' := printCafeTerm(T') .
```

- `printCafeTerm`, which prints a token or a singleton bubble:

```
op printCafeTerm : Term -> QidList .
eq printCafeTerm('token[T]) = downQid(T) .
eq printCafeTerm('CafeToken[T]) = downQid(T) .
eq printCafeTerm('CafeBubble[T]) = downQid(T) .
```

- `printCafeSort`, which just prints a token or a quoted identifier:

```
op printCafeSort : Term -> QidList .
eq printCafeSort('CafeToken[T]) = downQid(T) .
eq printCafeSort(T) = downQid(T) [owise] .
```

We define the sort `PrintCafePair` to return a pair consisting of the list of quoted identifier computed thus far and the set of variables defined in the module:

```
sort PrintCafePair .
op <_,_> : QidList OpDeclSet -> PrintCafePair [ctor] .
```

We also define methods `first` and `second` to obtain the corresponding components:

```
op first : PrintCafePair -> QidList .
eq first(< QIL, ODS >) = QIL .

op second : PrintCafePair -> OpDeclSet .
eq second(< QIL, ODS >) = ODS .
```

The function `printCafeBody*` receives the term standing for the original CafeOBJ specification, the obtained Maude module, the current database, a set of variables, and a list of parameters and returns a term of sort containing the representation of the module and the whole set of variables. It just traverses all the sentences in the module applying `printCafeBody` to each of them:

```
op printCafeBody* : Term Module Database OpDeclSet QidList -> PrintCafePair .
ceq printCafeBody*('__[T, T'], M, DB, ODS, PL) = <
                   if QIL =/= nil
                   then '\t QIL '\n
                   else nil
                   fi QIL', ODS'' >
 if < QIL, ODS' > := printCafeBody(T, M, DB, ODS, PL) /\
    < QIL', ODS'' > := printCafeBody*(T', M, DB, ODS', PL) .
ceq printCafeBody*(T, M, DB, ODS, PL) = < '\t QIL, ODS' >
 if < QIL, ODS' > := printCafeBody(T, M, DB, ODS, PL) .
```

The function `printCafeBody` receives a specific CafeOBJ statement and prints it. When dealing with importations, we just use the `printCafeName` shown above:

```
op printCafeBody : Term Module Database OpDeclSet QidList -> PrintCafePair .
ceq printCafeBody('protecting`(_`)[T], M, DB, ODS, PL) =
                                                < 'protecting '`( QIL '`), ODS >
 if QIL := printCafeName(DB, M, T) .
ceq printCafeBody('pr`(_`)[T], M, DB, ODS, PL) = < 'pr '`( QIL '`), ODS >
 if QIL := printCafeName(DB, M, T) .
ceq printCafeBody('extending`(_`)[T], M, DB, ODS, PL) =
                                                < 'extending '`( QIL '`), ODS >
 if QIL := printCafeName(DB, M, T) .
ceq printCafeBody('ex`(_`)[T], M, DB, ODS, PL) = < 'ex '`( QIL '`), ODS >
 if QIL := printCafeName(DB, M, T) .
ceq printCafeBody('including`(_`)[T], M, DB, ODS, PL) =
                                                < 'including '`( QIL '`), ODS >
 if QIL := printCafeName(DB, M, T) .
ceq printCafeBody('inc`(_`)[T], M, DB, ODS, PL) = < 'inc '`( QIL '`), ODS >
 if QIL := printCafeName(DB, M, T) .
ceq printCafeBody('using`(_`)[T], M, DB, ODS, PL) = < 'using '`( QIL '`), ODS >
 if QIL := printCafeName(DB, M, T) .
ceq printCafeBody('us`(_`)[T], M, DB, ODS, PL) = < 'us '`( QIL '`), ODS >
 if QIL := printCafeName(DB, M, T) .
```

Printing sorts requires to modify them in order to qualify the terms following the CafeOBJ syntax:

```
ceq printCafeBody('*`[_`]*[T], M, DB, ODS, PL) = < '* '`[ QIL '`] '*, ODS >
 if QIL := prettyprintParams(printCafeSort(T), PL) .
ceq printCafeBody('`[_`]['__[T, T']], M, DB, ODS, PL) = < '`[ QIL QIL' '`], ODS >
 if QIL := prettyprintParams(printCafeSort(T), PL) /\
    QIL' := prettyprintParams*(printCafeSortList(T'), PL) .
ceq printCafeBody('`[_`]['CafeToken[T]], M, DB, ODS, PL) = < '`[ QIL '`], ODS >
 if QIL := prettyprintParams(printCafeSort(T), PL) .
ceq printCafeBody('`[_`]['_<_[T, T']], M, DB, ODS, PL) = < '`[ QIL '< QIL' '`], ODS >
 if QIL := prettyprintParams*(printCafeSortList(T), PL) /\
    QIL' := prettyprintParams*(printCafeSortList(T'), PL) .
```

The auxiliary function required by this function are:

- `prettyprintParams`, which translates sorts from Maude syntax to CafeOBJ syntax. As we decribed in Section 1.2, we distinguish whether the character stands for a variable or a constant. If it is a constant, we have to reorder the term:

  ```
  op prettyprintParams : Qid QidList -> Qid .
  ceq prettyprintParams(Q, PL) = QI
   if St := string(Q) /\
      find(St, ":", 0) == notFound /\
      N := find(St, "$", 0) /\
      St' := substr(St, 0, N) /\
      Q' := qid(St') /\
      Q' in PL /\
      St'' := substr(St, N + 1, length(St)) /\
      QI := qid(St'' + "." + St') .
  ```

  If it is a variable, we have to mantain the variable name at the beginning of the character:

  ```
  ceq prettyprintParams(Q, PL) = QI
   if St := string(Q) /\
      N := find(St, ":", 0) /\
  ```

```
      N' := find(St, "$", 0) /\
      St' := substr(St, 0, N) /\
      St'' := substr(St, N + 1, sd(N', N + 1)) /\
      Q' := qid(St'') /\
      Q' in PL /\
      St''' := substr(St, N' + 1, length(St)) /\
      QI := qid(St' + ":" + St''' + "." + St'') .
  eq prettyprintParams(Q, PL) = Q [owise] .
```

and prettyprintParams* just traverses the list, applying prettyprintParams to each element:

```
  op prettyprintParams* : QidList QidList -> QidList .
  eq prettyprintParams*(nil, PL) = nil .
  eq prettyprintParams*(Q QIL, PL) =
               prettyprintParams(Q, PL)
               prettyprintParams*(QIL, PL) .
```

- printCafeSortList, which prints all the subterms of the given term by traversing the flattened list:

```
  op printCafeSortList : Term -> QidList .
  ceq printCafeSortList('__[TL]) = QIL
   if QIL := printCafeSortList*(flatten(TL)) .
  eq printCafeSortList(T) = printCafeSort(T) [owise] .
```

  where printCafeSortList* just traverses the list, applying printCafeSort to each element:

```
  op printCafeSortList* : TermList -> QidList .
  eq printCafeSortList*(empty) = nil .
  eq printCafeSortList*((T, TL)) = printCafeSort(T) printCafeSortList*(TL) .
```

  and flatten just removes the juxtaposition operator from a list:

```
  op flatten : TermList -> TermList .
  eq flatten(empty) = empty .
  eq flatten(('__[TL], TL')) = flatten((TL, TL')) .
  eq flatten((T, TL)) = T, flatten(TL) [owise] .
```

The printing function does not print variables, since the parsing to compute the structure of the terms eliminates the syntactic sugar. Instead, we add the variables to the set of operators:

```
ceq printCafeBody('var_:_.['neCafeTokenList[T], 'CafeToken[T']], M, DB, ODS, PL) =
                                                  < nil, ODS ODS' >
 if ODS' := parseVars(downQidList(T), downQid(T')) .
ceq printCafeBody('vars_:_.['neCafeTokenList[T], 'CafeToken[T']], M, DB, ODS, PL) =
                                                  < nil, ODS ODS' >
 if ODS' := parseVars(downQidList(T), downQid(T')) .
```

Although several cases are distinguished for printing operators, most of them work in the same way. We just show the case for basic operator declarations, where the head and the coarity are printed:

```
ceq printCafeBody('op_:`->_.[T, 'CafeToken[T']], M, DB, ODS, PL) =
                                 < 'op QIL ': '-> Q', ODS >
 if QIL := printCafeOperatorHead(T) /\
    Q' := downQid(T') .
```

and for predicates, where we take care of parameters in the arity:

```
ceq printCafeBody('pred_:_.[T, T'], M, DB, ODS, PL) =
                                 < 'pred QIL ': QIL' '., ODS >
 if QIL := printCafeOperatorHead(T) /\
    QIL' := prettyprintParams*(printCafeSortList(T'), PL) .
```

The auxiliary function `printCafeOperatorHead` just puts together all the characters in the operator name:

```
op printCafeOperatorHead : Term -> QidList .
eq printCafeOperatorHead('CafeToken[T]) = downQid(T) .
ceq printCafeOperatorHead('__[T, T']) = QIL QIL'
 if QIL := printCafeOperatorHead(T) /\
    QIL' := printCafeOperatorHead(T') .
```

The printing function for equations and rules are also very similar, so we will only describe an unconditional equation and a conditional rule. Since we are interested in each part of the equation the parsing in this case is complicated, so we explain it in detail:

- We check whether the term has a label by parsing the term after trying to extract it. If the parsing succeeds, then we keep in the `TW` the term after removing the label; otherwise, we keep the same term.
- Similarly, we extract the attributes from the righthand side. However, this function cannot fail, so it is not necessary to parse the obtained term.
- We compute the list of quoted identifiers standing for the new lefthand side. It will be use to compute the new variables, kept in `ODS'`.
- We solve the bubbles in the lefthand side and keep the result in `T1`.
- We solve the bubbles in the righthand side and keep the obtained term in `T2`. Solving this term requires a special function that takes into account the new variables that appeared in the lefthand side.
- The representation of these terms is kept in `QIL'` and `QIL''`, respectively.
- The representation of the label is stored in `QILL`, while the representation of the attributes is kept in `QILA`.
- Finally, the equation is built and returned.

```
ceq printCafeBody('eq_=_.['CafeBubble[T], 'CafeBubble[T']], M, DB, ODS, PL) =
                       < 'eq QILL QIL' '= QIL'' QILA '., ODS >
 if TW := if solveBubbles('bubble[extractLabel(T)], M, false, ODS, DB) :: Term
          then extractLabel(T)
          else T
          fi /\
    TW' := removeEqAtS(T') /\
    QIL := downQidList(TW) /\
    ODS' := ODS opDeclSetFromQidList(QIL) /\
    T1 := solveBubbles('bubble[TW], M, false, ODS, DB) /\
```

```
   T2 := solveSecondTerm(M, 'bubble[TW], 'bubble[TW'], ODS', DB) /\
   QIL' := prettyprintParams*(addSpace(metaPrettyPrint(M, T1, printOpts)), PL) /\
   QIL'' := prettyprintParams*(addSpace(metaPrettyPrint(M, T2, printOpts)), PL) /\
   QILL := printLabel(T) /\
   QILA := printAtS(getEqAtS(T')) .
```

Regarding conditional transitions, we have to extended the operations we performed for unconditional statements by:

- Extracting the attributes from the term standing for the condition, since this is now the last term.
- Extending the module with information about sorts, required by the operators defined with the sort `Universal`, to parse the condition. This is performed by applying the Full Maude function `addInfoConds` to the module extended with the variables defined in the lefthand side.
- Using this extended module to solve the bubbles in the condition, and then printing it in `QIL3`.
- The printed transition is finally composed and returned.

```
ceq printCafeBody('ctrns_=>_if_.['CafeBubble[T], 'CafeBubble[T'],
                 'CafeBubble[T'']], M, DB, ODS, PL) =
                        < 'ctrns QILL QIL' '=> QIL'' 'if QIL3 QILA '., ODS >
 if TW := if solveBubbles('bubble[extractLabel(T)], M, false, ODS, DB) :: Term
         then extractLabel(T)
         else T
         fi /\
    TW' := removeEqAtS(T'') /\
    QIL := downQidList(TW) /\
    ODS' := ODS opDeclSetFromQidList(QIL) /\
    T1 := solveBubbles('bubble[TW], M, false, ODS, DB) /\
    T2 := solveSecondTerm(M, 'bubble[TW], 'bubble[T'], ODS', DB) /\
    QIL' := prettyprintParams*(addSpace(metaPrettyPrint(M, T1, printOpts)), PL) /\
    QIL'' := prettyprintParams*(addSpace(metaPrettyPrint(M, T2, printOpts)), PL) /\
    M' := addInfoConds(addOps(ODS', M)) /\
    QIL3 := prettyprintParams*(printCond(M,
              solveBubblesCond('bubble[TW'], M, M', false, ODS', DB)), PL) /\
    QILL := printLabel(T) /\
    QILA := printAtS(getEqAtS(T'')) .
```

The auxiliary functions used in this case are:

- `solveSecondTerm`, which add a special operator `_=_` on a new sort `@@@`. We then solve the bubbles in this new sort by using the new operator. Finally, the constants used in the parsing process are transformed back into variables if they appear in the operator set:

  ```
  op solveSecondTerm : Module Term Term OpDeclSet Database -> Term .
  ceq solveSecondTerm(M, 'bubble[T], 'bubble[T'], ODS, DB) = T2
   if M' := addOps((ODS op '_=_ : 'Universal 'Universal -> '@@@ [poly(1 2)] .),
             addSorts('@@@, M)) /\
      QIL := downQidList(T) /\
      QIL' := downQidList(T') /\
      RP := metaParse(M', '`( QIL '`) '= '`( QIL' '`), '@@@) /\
      '_=_[T1, T2] := constsToVars(getTerm(RP), ODS) .
  ```

- `extractLabel`, which extracts a label at the beggining of the term:

```
op extractLabel : Term -> Term .
ceq extractLabel('__[''`[.Qid, T, ''`].Qid, '':.Qid, TL]) = '__[TL]
 if TL =/= empty /\
    Q := downTerm(T) .
eq extractLabel(T) = T [owise] .
```

- printLabel, which transforms a term into a list of quoted identifiers. If the term does not correspond with a label, it is not printed:

```
op printLabel : Term -> QidList .
ceq printLabel('__[''`[.Qid, T, ''`].Qid, '':.Qid, TL]) = ' '`[ Q '`] ' ':
 if TL =/= empty /\
    Q := downTerm(T) .
eq printLabel(T) = nil [owise] .
```

- removeEqAtS, which traverses the list of terms, looking for possible attributes:

```
op removeEqAtS : Term -> Term .
ceq removeEqAtS('__[TL]) = if sizeTL(TL') > 1
                           then '__[TL']
                           else TL'
                           fi
 if TL' := removeEqAts*(TL) .
eq removeEqAtS(TL) = TL [owise] .
```

where removeEqAts* removes the nonexec and metadata attributes:

```
op removeEqAts* : TermList -> TermList .
eq removeEqAts*((TL, ''`{.Qid, ''nonexec.Qid, TL'', ''`}.Qid)) = TL .
eq removeEqAts*((TL, ''`{.Qid, ''metadata.Qid, TL'', ''`}.Qid)) = TL .
eq removeEqAts*(TL) = TL [owise] .
```

and sizeTL just computes the size of a list of terms:

```
op sizeTL : TermList -> Nat .
eq sizeTL(empty) = 0 .
eq sizeTL((T, TL)) = s(sizeTL(TL)) .
```

- Analogously, getEqAtS returns the terms standing for the attributes:

```
op getEqAtS : Term -> TermList .
eq getEqAtS('__[TL]) = getEqAtS*(TL) .
eq getEqAtS(T) = empty [owise] .
```

where getEqAtS* just looks for the nonexec or metadata attributes:

```
op getEqAtS* : Term -> TermList .
eq getEqAtS*((TL, ''`{.Qid, ''nonexec.Qid, TL'', ''`}.Qid)) =
                                ''`{.Qid, ''nonexec.Qid, TL'', ''`}.Qid .
eq getEqAtS*((TL, ''`{.Qid, ''metadata.Qid, TL'', ''`}.Qid)) =
                                ''`{.Qid, ''metadata.Qid, TL'', ''`}.Qid .
eq getEqAtS*(TL) = empty [owise] .
```

- Finally, `printAtS` prints the list by placing spaces at both sides (since the attributes are placed inside curly braces):

```
op printAtS : TermList -> QidList .
eq printAtS(empty) = nil .
eq printAtS(TL) = ' downQidList(TL) ' [owise] .
```

```
endm
```

1.4 Defining commands for CafeOBJ specifications

We present here how to define the behavior of the commands specified for CafeOBJ specifications. To add any other command the programmer must define it in the module `TRANSLATION-COMMANDS` described in Section 1.1 and then define its behavior in the `CAFE2MAUDE-DATABASE-HANDLING` module:

```
mod CAFE2MAUDE-DATABASE-HANDLING is
  pr CAFE-PRETTY-PRINT .
  pr CafePARSER .
```

This module define the `CafeDatabaseClass` sort, which will be used in all the rules involving CafeOBJ specifications. Since we also want the rest of rules from Full Maude to work, we add a subsort declaration stating that our class is a subclass of `DatabaseClass`, defined in Full Maude. Finally, we define a constant `CafeDatabase` for creating new objects:

```
  sort CafeDatabaseClass .
  subsort CafeDatabaseClass < DatabaseClass .
  op CafeDatabase : -> CafeDatabaseClass [ctor] .
```

We also define a new attribute, that will store whether the user wants the system to perform a strict translation:

```
  op strict :_ : Bool -> Attribute [ctor] .
```

The rule `load-CafeLOOSE` is in charge of loading a module with loose semantics. It uses the function `procCafeMod` from Section 1.2 to parse the terms. If there is no errors (i.e., the variable `QIL` is equals to `nil`) then the database is updated and a message indicating that the module has been introduced is shown. If `QIL` we check whether it contains an error that can be solved by translating theories as modules. If this is the case and the user does not need a strict translation (the boolean variable `B` in the attribute `strict` is set to `false`) then the database is updated and a warning message is shown. Otherwise, the database is not modified and a message is printed:

```
  crl [load-CafeLOOSE] :
      < O : X@Database | input : ('cmod*_`{_`}[T, T']), db : DB, output : nil,
                         default : ME, strict : B, Atts >
   => if QIL == nil
      then
      < O : X@Database | input : nilTermList, db : DB',
                         output : ('Introduced 'Loose 'Cafe 'Module:
                                   '\g header2Qid(parseHeader(getHeader(T))) '\o '\n),
                         default : parseHeader(getHeader(T)), strict : B, Atts >
      else if QIL == paramThWarn
           then if B then
                    < O : X@Database | input : nilTermList, db : DB, output : QIL,
                                       default : ME, strict : B, Atts >
```

```
            else
                 < O : X@Database | input : nilTermList, db : DB', output : QIL,
                                    default : parseHeader(getHeader(T)),
                                    strict : B, Atts >
            fi
        else
            < O : X@Database | input : nilTermList, db : DB, output : QIL,
                               default : ME, strict : B, Atts >
        fi
    fi
 if < DB', QIL > := procCafeMod('cmod*_`{_`}[T, T'], DB) .
```

The auxiliary function `getHeader` just returns the module name without parameters and using a Maude token constructor:

```
op getHeader : Term -> Term .
eq getHeader('CafeToken[T]) = 'token[T] .
eq getHeader('_`(_`)['CafeToken[T], T']) = 'token[T] .
```

Similarly to the previous rule, `load-CafeTIGHT` is in charge of loading modules with tight semantics. In this case we do not have to take into account whether the message contains a warning message, because parameterized modules are allowed in Maude. Hence, we just parse the terms with `procCafeMod` and update the database if no errors are found:

```
 crl [load-CafeTIGHT] :
     < O : X@Database | input : ('cmod!_`{_`}[T, T']), db : DB, output : nil,
                        default : ME, strict : B, Atts >
  => if QIL == nil
     then
     < O : X@Database | input : nilTermList, db : DB',
                        output : ('Introduced 'Tight 'Cafe 'Module: '\g
                              header2Qid(parseHeader(getHeader(T))) '\o '\n),
                        default : parseHeader(getHeader(T)), strict : B, Atts >
     else
     < O : X@Database | input : nilTermList, db : DB, output : QIL,
                        default : ME, strict : B, Atts >
     fi
  if < DB', QIL > := procCafeMod('cmod!_`{_`}[T, T'], DB) .
```

Finally, the rule `load-CafeVIEW` loads a CafeOBJ view into the database. It uses the `procCafeView` function to parse the view and, if no errors are found, then the database is updated:

```
crl [load-CafeVIEW] :
    < O : X@Database | input : ('cview_from_to_`{_`}[T, TL]), db : DB,
                       output : nil, Atts >
 => if QIL == nil
    then
    < O : X@Database | input : nilTermList, db : DB',
                       output : ('Introduced 'Cafe 'View: '\g
                                 header2Qid(parseHeader(getHeader(T))) '\o '\n),
                       Atts >
    else
    < O : X@Database | input : nilTermList, db : DB, output : QIL, Atts >
    fi
 if < DB', QIL > := procCafeView('cview_from_to_`{_`}[T, TL], DB) .
```

The rule original-cafe-module displays the CafeOBJ module originally introduced by the user. It just obtains the module name from the command, looks for the module in the database and prints it with printCafeModule:

```
crl [original-cafe-module] :
    < O : X@Database | input : ('original`CafeOBJ`module_.['token[T]]),
                       output : nil, db : DB, Atts >
 => < O : X@Database | input : nilTermList, output : QIL,
                       db : DB, Atts >
 if Q := downQid(T) /\
    < T' ; ODS ; M > := getTermModule(Q, DB) /\
    M' := getFlatModule(Q, DB) /\
    QIL := printCafeModule(T', M', DB, false) .
```

Alternatively, the rule original-cafe-module-error is applied when the module cannot be found:

```
crl [original-cafe-module-error] :
    < O : X@Database | input : ('original`CafeOBJ`module_.['token[T]]),
                       output : nil, db : DB, Atts >
 => < O : X@Database | input : nilTermList, output : ('\n '\r 'ERROR: '\o
                                         'The 'module 'cannot 'be 'found. '\n),
                       db : DB, Atts >
 if Q := downQid(T) /\
    not getTermModule(Q, DB) :: Tuple{Term,OpDeclSet,Module} .
```

The rule strict-on sets the value in strict to true, and prints a message indicating that the operation was successful:

```
rl [strict-on] :
   < O : X@Database | input : ('strict`translation`on`..@Command@), strict : B,
                      output : nil, Atts >
=> < O : X@Database | input : nilTermList, strict : true,
                      output : ('\n '\b 'The 'modules 'will 'be 'introduced
                                'as 'originally 'written. '\o '\n), Atts > .
```

Similarly, the rule strict-off sets the value in strict to false and prints the corresponding message:

```
rl [strict-off] :
   < O : X@Database | input : ('strict`translation`off`..@Command@), strict : B,
                      output : nil, Atts >
=> < O : X@Database | input : nilTermList, strict : false,
                      output : ('\n '\b 'The 'translation 'will 'adapt 'CafeOBJ
                                'specifications 'to 'meet 'Maude 'requirements 'when
                                'possible. '\o '\n), Atts > .
endm
```

The module LOOP-PRE-PROCESSING performs some normalization actions on the modules to simplify the parsing functions and the use of the metaParse command. This is specially important due to the use of bubbles, that do not delimit the terms. Hence the functions in this module:

- Add a dot at the end of the statements that do not require it and are not "closed" by themselves (e.g. the sort declaration constructor [_] is closed, while view mappings are not closed).

- Modify some characters that might cause ambiguity, such as the `mod` keyword at the beginning of a module and is also used by Maude modules.

Although the functions distinguish several cases all of them are basically implemented in the same way, so we do not show the details here:

```
mod LOOP-PRE-PROCESSING is
  pr LOOP-MODE .
  pr EXT-BOOL .

  ...
endm
```

Finally, the `CAFE2MAUDE` module is the standard module dealing with input/output through the Loop Mode [1, Chapter 17]. Basically, this module uses a tuple built with the operator `[_,_,_]`, where the first argument corresponds to the input introduced by the user, the third one the output shown to the user, and the second one is a term of sort `State` that can be defined by the user for each application:

```
mod CAFE2MAUDE is
  ex LOOP-PRE-PROCESSING .
  pr META-CAFE2MAUDE-SIGNATURE .
  pr CAFE2MAUDE-DATABASE-HANDLING .
```

We will use the sort `Object` for the current state, which means that we will store the values in a term built with the operator `<_:_|_>`, with the first argument the name of the object, the second one the name of the class, and the third one a set of attributes. We also define a constant `o` of sort `Oid` to define the name of the initial state:

```
  subsort Object < State .
  op o : -> Oid .
```

We define the constant `cafe2maude-init` as the initial system. The user has to rewrite it with the special command `loop` to start the input/output loop.

```
  op cafe2maude-init : -> System .
```

When the user types that command the system will apply the rule `init` below. It creates the whole sytem, with an empty list of quoted identifiers in the input (the first `nil`), another one in the output (the second `nil`), and an object with name `o`, class `CafeDatabase`, and attributes for the database (verb"db"), the parsed `input`, the messages we want to print (`output`), the `default` modules (all these attributes are inherited from Full Maude), and for indicating whether the translation is `strict`, which is initially set to `false`:

```
  rl [init] :
     cafe2maude-init
  => [nil, < o : CafeDatabase | db : initialDatabase, input : nilTermList,
                                output : ('\n '\t '\b 'CafeOBJ2Maude '1.0 '\o
                                          'started. '\n 'CafeOBJ 'specifications
                                          'can 'be 'introduced 'now 'into 'the
                                          'Full 'Maude 'database. '\n),
                                 default : 'CONVERSION, strict : false >, nil] .
```

The rule `input` moves the list of quoted identifiers in the first argument of the tuple to the `input` attribute, trying to parse it first. We use the `CafeGRAMMAR` module to parse this input since this module, as explained in Section 1.1, contains the syntax of all our programs and commands, as well as the syntax inherited from Full Maude:

```
  rl [input] :
     [QI QIL, < O : X@Database | input : nilTermList, output : nil, Atts >, QIL' ]
  => if metaParse(CafeGRAMMAR, QI QIL, '@Input@) :: ResultPair
     then [nil,
           < O : X@Database | input : getTerm(metaParse(CafeGRAMMAR, QI QIL, '@Input@)),
                              output : nil, Atts >,
          QIL']
     else [nil,
           < O : X@Database | input : nilTermList,
                              output : ('\r 'WARNING: '\o printSyntaxError(metaParse(
                                         CafeGRAMMAR, QI QIL, '@Input@), QI QIL) '\n
                                         'ERROR: 'No 'parse 'for 'input.), Atts >,
           QIL' ]
     fi .
```

On the other hand, the rule `output` moves the output from the attribute `output` to the third element of the tuple:

```
  rl [output] :
     [QIL, < O : X@Database | output : (QI QIL'), Atts >, QIL'']
  => [QIL, < O : X@Database | output : nil, Atts >, (QI QIL' QIL'')] .
endm
```

2 Integrating CafeOBJ with the Constructor-based Inductive Theorem Prover

We present in this section how to integrate the specification shown in the previous version with any existing tool for Maude specifications implemented in Full Maude.

2.1 Extending the tool

We first present the new modules required to process the commands from the tool we want to extend. Note that this part is different for each tool. The module `CAFE-CITP-COMMANDS-PROC` is in charge of processing the commands.

```
mod CAFE-CITP-COMMANDS-PROC is
  pr PROVE-COMMANDS-PROC .
  pr CAFE-PRETTY-PRINT .
```

The function `procGoalsCafe` is in charge of processing the initial goal introduced by the user. It receives the current database, the module expression standing for the module where the proof will take place, and the term containing the goal sentences and returns a term of sort `ProveResult`, composed of the updated database, the generated proof tree, and a list of quoted identifiers propagating errors. If the parsing of the sentences fails, we do not modify the database, the proof tree is `null`, and an error message is sent:

```
  op procGoalsCafe : Database ModuleExpression Term -> ProveResult .
  ceq procGoalsCafe(DB, ME, T) = << DB ; null ; ('\r 'ERROR: '\o
                                                  'bad 'goal 'specified) >>
   if sentErr(QIL) := parseSentenceCafe(T) .
```

Otherwise, the `procGoalCmdCafe` is applied:

```
  eq procGoalsCafe(DB, ME, T) = procGoalCmdCafe(DB, ME, parseSentenceCafe(T)) [owise] .
```

The auxiliary function `parseSentenceCafe` takes a term as argument and returns a set of sentences, a special sort defined by the CITP to put together equations and rules. Thus, we translate equations into equations and transitions into rules:

```
op parseSentenceCafe : Term -> SentenceSet .
eq parseSentenceCafe('eq_=_;[T, T']) = (eq T = T' [none]) .
eq parseSentenceCafe('ceq_=_if_;[T, T', T'']) =
                              (ceq T = T' if T'' = 'true.Bool [none]) .
eq parseSentenceCafe('trans_=>_;[T, T']) = (rl T => T' [none]) .
eq parseSentenceCafe('trns_=>_;[T, T']) = (rl T => T' [none]) .
eq parseSentenceCafe('ctrans_=>_if_;[T, T', T'']) =
                              (crl T => T' if T'' = 'true.Bool [none]) .
eq parseSentenceCafe('ctrns_=>_if_;[T, T', T'']) =
                              (crl T => T' if T'' = 'true.Bool [none]) .
ceq parseSentenceCafe('__[T, T']) = Snt ScS
 if Snt := parseSentenceCafe(T) /\
    ScS := parseSentenceCafe(T') .
eq parseSentenceCafe(Q[TL]) = sentErr('Bad 'goal)  [owise] .
```

The parsing continues with `procGoalCmdCafe`. This function extracts the module from the database and tries to solve the bubbles. If this parsing returns an error, it is returned to the user, keeping the same database and building the `null` proof tree:

```
op procGoalCmdCafe : Database ModuleExpression SentenceSet -> ProveResult .
ceq procGoalCmdCafe(DB, ME, ScS) = << DB ; null ; QIL >>
 if M := getFlatModule(ME, DB) /\
    sentErr(QIL) := solveBubblesCafe(ScS, M, addInfoConds(M), false,
                                     getVars(ME, DB), DB) .
```

If the sentences are successfully parsed, the new goal is created with them:

```
ceq procGoalCmdCafe(DB, ME, ScS) = setGoal(DB, M, ScS')
 if M := getFlatModule(ME, DB) /\
    ScS' := solveBubblesCafe(ScS, M, addInfoConds(M), false, getVars(ME, DB), DB) .
```

Ohterwise, we return a module error:

```
eq procGoalCmdCafe(DB, ME, ScS) = << DB ; null ; ('\r 'ERROR: '\o 'Module
                                                 'error.) >> [owise] .
```

The auxiliary function `solveBubblesCafe` is in charge of building the terms defined inside bubbles in equations and rules. Since all the equations for this function are very similar, we only show how they are solved for an equation. In this case, we only need to extend the set of operators standing for the variables, and use it with the `solveBubbles` function from Full Maude:

```
op solveBubblesCafe : SentenceSet Module Module Bool OpDeclSet Database
                      ~> SentenceSet .
ceq solveBubblesCafe((eq 'bubble[T] = T' [none]), M, M', B, VDS, DB) =
                               eq T1 = T2 [AtS]
 if QIL := downQidList(T) /\
    ODS := VDS opDeclSetFromQidList(QIL) /\
    eq T1 = T2 [AtS] . := solveBubbles(eq 'bubble[T] = T' [none] .,
                                       M, M', B, ODS, DB) .
```

Moreover, we also define the equations for dealing with several sentences and with the empty set of sentences (`none`):

```
ceq solveBubblesCafe(Snt ScS, M, M', B, VDS, DB) =
                              solveBubblesCafe(Snt, M, M', B, VDS, DB)
                              solveBubblesCafe(ScS, M, M', B, VDS, DB)
 if ScS =/= none .
eq solveBubblesCafe(none, M, M', B, VDS, DB) = none .
```

The function `procInitLemmaCafe` just applies the general function `procInitLemma` from the CITP:

```
op procInitLemmaCafe : Database PTree Qid QidList -> ProveResult .
eq procInitLemmaCafe(DB, P, Q, QIL) = procInitLemma(DB, P, Q, QIL) .
```

We also define several printing functions to print sentences and goals in CafeOBJ style. We present some of them:

- Printing the empty set of sentences returns `nil`:

  ```
  op printSentencesCafe : Module SentenceSet -> QidList .
  eq printSentencesCafe(M, none) = nil .
  ```

- Equations are printed by printing each side, the labels and the attributes:

  ```
  ceq printSentencesCafe(M, (eq T = T' [AtS]) ScS) =
                  ('\n '\t '\s '\s '\g 'eq QILL '\s '\o QIL1
                   '\n '\t '\s '\s '\s '\s '\g '= '\s '\o QIL2
                   '\s QIL3 '.
                   printSentencesCafe(M, ScS))
   if QIL1 := eMetaPrettyPrint(M, T) /\
      QIL2 := eMetaPrettyPrint(M, T') /\
      QIL3 := printCafeAtS(AtS) /\
      QILL := printLabel(AtS) .
  ```

- In some cases the prover adds extra sentences, that we want to print in a special way. If the option asking to print the module is set to `false` nothing is printed, but in other case we print these extra sentences:

  ```
  op printExtraStmnts : Module Bool -> QidList .
  eq printExtraStmnts(FM, false) = nil .
  eq printExtraStmnts(FM, true) =
            printAddedEqs(FM, getAddedEqs(getEqs(FM)))
            printAddedRls(FM, getAddedRls(getRls(FM)))
            printAddedEqLemmas(FM, getEqLemmas(getEqs(FM)))
            printAddedRlLemmas(FM, getRlLemmas(getRls(FM))) .
  ```

- When printing rule lemmas, besides printing both sides, the label, and the attributes, we show an extra line indicating that it is a lemma added by the tool. Note that we print rules using CafeOBJ syntax for transitions:

  ```
  op printAddedRlLemmas : Module RuleSet -> QidList .
  ceq printAddedRlLemmas(M, rl T => T' [AtS] . RS) =
                  ('\n '\t '\s '\s '\g 'trans QILL '\s '\o QIL1
                   '\n '\t '\s '\s '\s '\s '\g '=> '\s '\o QIL2
                   '\s QIL3 '. '\b
                   ' '--> 'Lemma 'added 'by 'the 'prover. '\o
                   printAddedRlLemmas(M, RS))
   if QIL1 := eMetaPrettyPrint(M, T) /\
      QIL2 := eMetaPrettyPrint(M, T') /\
      QIL3 := printCafeAtS(AtS) /\
      QILL := printLabel(AtS) .
  ```

- Modules are printed by combining the functions for extra statements shown above and the printing functions from Section 1.3:

```
op printCafeModuleGoal : Term Module Database Bool -> QidList .
ceq printCafeModuleGoal('cmod!_`{_`}[T, T'], M, DB, B) =
            '\n 'mod! printCafeName(DB, M, T) '`{
            '\n first(printCafeBody*(paramSortsMap(T', PL), M, DB, none, PL))
            printExtraStmnts(M, B)
            '\n '`}
 if PL := paramNames(T) .
ceq printCafeModuleGoal('cmod*_`{_`}[T, T'], M, DB, B) =
            '\n 'mod* printCafeName(DB, M, T) '`{
            '\n first(printCafeBody*(paramSortsMap(T', PL), M, DB, none, PL))
            printExtraStmnts(M, B)
            '\n '`}
 if PL := paramNames(T) .
```

- The proof tree is printed by printing the current goal:

```
op prettyPrintProofTreeCafe : PTree Database Bool Term -> QidList .
eq prettyPrintProofTreeCafe([ Q ; G ; B' ; PS' ], DB, B, T) =
'========================== '\s 'GOAL Q '========================== '\n
(prettyPrintGoalCafe(G, DB, B, T)  '\b  (if B'
                                       then 'proved
                                       else 'unproved
                                       fi ) '\o '\n ) .
```

- Finally, to print the goal we first check whether the whole module must be displayed. In this case we print it, otherwise only the extra statements are shown. Finally, the sentences composing the goal are shown:

```
  op prettyPrintGoalCafe : Goal Database Bool Term -> QidList .
  eq prettyPrintGoalCafe(< FM, ScS, L >, DB, B, T) =
     ('< (if B then printCafeModuleGoal(T, FM, DB, false)
          else ('Module '\g getName(FM) '\o 'is 'concealed '\n '... '\n
                printAddedEqs(FM, getAddedEqs(getEqs(FM)))
                printAddedRls(FM, getAddedRls(getRls(FM)))
                printAddedEqLemmas(FM, getEqLemmas(getEqs(FM)))
                printAddedRlLemmas(FM, getRlLemmas(getRls(FM)))
               '\b 'end '\o '\n) fi ) '`, printSentencesCafe(FM, ScS) '>  '\n ) .
  eq prettyPrintGoalCafe(G, DB, B, T) = ('Bad 'GOAL) [owise]  .
endm
```

The module CAFE-UI is in charge of implementing the rules required to execute the CITP commands for CafeOBJ specifications. We will only present the main rules and some examples of error handling:

```
mod CAFE-UI is
  pr CAFE2MAUDE-DATABASE-HANDLING .
  pr CAFE-CITP-COMMANDS-PROC .
  inc THM-DATABASE-HANDLING .
```

This module defines a new attribute `originalCafeModule` which is in charge of storing the original CafeOBJ specification used for the current goal. It is necessary to store it because the default module can be modified when introducing new modules during a proving session:

```
op originalCafeModule :_ : TermList -> Attribute [ctor] .
```

The goal-Mod-cafe rule processes a goal introduced by the user. It parses the module expression and looks for the module in the database. Once it is found, we check that it is a CafeOBJ specification with isCafeMod?. Then the sentences are parsed with the function procGoalsCafe shown above and printed with prettyPrintProofTreeCafe. It also sets the originalCafeModule attribute to the selected module. Note that we use an attribute language with value cafeobj. It will be described in Section 2.2, but its behavior is clear: state that this rule is only applied if CafeOBJ is the selected specification language.

```
crl [goal-Mod-cafe] :
    < O : X@Database | db : DB, input : ('goal_|-_[T, T']), output : nil,
                       default : ME, pTree : P, currentGoal : GID, showMod : B,
                       language : cafeobj, originalCafeModule : TL, Atts >
 => < O : X@Database | db : DB, input : nilTermList,
                       output : (if QIL == 'OK
                                 then QIL'
                                 else QIL
                                 fi), default : ME, pTree : P',
                       currentGoal : getDefaultGoalIndex(P'), showMod : B,
                       language : cafeobj, originalCafeModule : T1, Atts >
 if ME' := parseModExp(T) /\
    < DB' ; ME'' > := evalModExp(ME', DB) /\
    < T1 ; ODS ; M > := getTermModule(ME'', DB') /\
    isCafeMod?(T1) /\
    << DB'' ; P' ; QIL >> := procGoalsCafe(DB', ME'', T') /\
    QIL' := prettyPrintProofTreeCafe(P', DB', B, T1) '\n '\g
            'INFO: '\o 'an 'initial 'goal 'generated! .
```

where the function isCafeMod? just checks the operator at the top of the term:

```
op isCafeMod? : Term -> Bool .
eq isCafeMod?('cmod!_`{_`}[TL]) = true .
eq isCafeMod?('cmod*_`{_`}[TL]) = true .
eq isCafeMod?(T) = false [owise] .
```

When the selected module does not correspond to a CafeOBJ specification, we use the rule goal-Mod-cafe-error-module-type to display an error message:

```
crl [goal-Mod-cafe-error-module-type] :
    < O : X@Database | db : DB, input : ('goal_|-_[T, T']), output : nil,
                       language : cafeobj, Atts >
 => < O : X@Database | db : DB, input : nilTermList,
                       output : ('\r 'WARNING: '\o 'the 'selected 'module 'is 'not
                                 'a 'CafeOBJ 'specification.),
                       language : cafeobj, Atts >
 if ME' := parseModExp(T) /\
    < DB' ; ME'' > := evalModExp(ME', DB) /\
    < null ; ODS ; M > := getTermModule(ME'', DB') .
```

The showGoal-cafe rule displays the current goal by using the prettyPrintProofTreeCafe function:

```
rl [showGoal-cafe] :
   < O : X@Database | db : DB, input : ('show`goal`..@Command@),
                      output : nil , pTree : P, currentGoal : GID,
                      showMod : B, language : cafeobj,
```

```
                         originalCafeModule : T, Atts >
  => < O : X@Database | db : DB, input : nilTermList, currentGoal : GID,
                         output : if GID =/= 'nil
                                  then
                                    prettyPrintProofTreeCafe(getPTree(P, GID), DB, B, T)
                                  else
                                    ('\r 'WARNING: '\o 'no 'goals 'to 'show!)
                                  fi, pTree : P, showMod : B,
                         language : cafeobj, originalCafeModule : T, Atts > .
```

The showGoal-id-cafe rule displays the goal selected by the user by applying the function prettyPrintProofTreeCafe:

```
  crl [showGoal-id-cafe] :
      < O : X@Database | db : DB, input : ('show`goal_.['token[T]]),
                          output : nil, pTree : P, currentGoal : GID,
                          showMod : B, language : cafeobj,
                          originalCafeModule : T', Atts >
   => < O : X@Database | db : DB, input : nilTermList, currentGoal : downQidList(T),
                          output : prettyPrintProofTreeCafe(P', DB, B, T'),
                          pTree : P, showMod : B, language : cafeobj,
                          originalCafeModule : T', Atts >
   if P' := getPTree(P, downQidList(T)) .
```

The applyRule-cafe-not-finish rule applies the given rule to the default goal. It first checks whether it is a valid rule. If true, then it is applied to the current goal and the new set of goals is displayed:

```
  crl [applyRule-cafe-not-finished] :
      < O : X@Database | db : DB, input : ('apply_.['bubble[T]]), output : nil,
                          pTree : P, currentGoal : GID, tactic : N, showMod : B,
                          language : cafeobj, originalCafeModule : T', Atts >
   => < O : X@Database | db : DB, input   : nilTermList,
                          output  : ('~~~~~~~~~~~~~~~~~~~~~~~ '\s '\s '\s '\s '\s '\s
                                     'Generated 'GOALS '\s '\s '\s '\s '\s '\s
                                     '~~~~~~~~~~~~~~~~~~~~~~~ '\n '\n
                                     prettyPrintProofTreeAuxCafe((P'' PS), DB, B, T')
                                     '\n '\g 'INFO: '\o (qid(string(num(P'' PS), 10))
                                     'goal`(s`) 'generated! '\n '\g 'INFO: '\o 'Next
                                     'goal 'to 'be 'proved 'is '\r
                                     getDefaultGoalIndex(P'') '\o )),
                          pTree : addPTreeSet(P, GID, (P'' PS)),
                          currentGoal : getDefaultGoalIndex(P''), tactic : N,
                          showMod : B, language : cafeobj, originalCafeModule : T',
                          Atts >
   if isValidRule(downQidList(T)) /\
      P' := getPTree(P, GID) /\
      (P'' PS) := applyRules(downQidList(T), P') .
```

Finally, we define a constant with the initial values of the attributes related to CafeOBJ specifications:

```
  op initCafeAttS : -> AttributeSet .
  eq initCafeAttS = strict : false, originalCafeModule : empty .
endm
```

2.2 Modifications required in the CITP

We present here the modifications that must be performed into the tool to support CafeOBJ specifications, as well as to add the new commands defined in the previous sections. We present here the modifications for the Constructor-based Inductive Theorem Prover, but they are general and can be applied to any other tool. Moreover, note that this modifications are performed *in the interface*, while the tool itself remains unchanged.

First, we need to distinguish the language we are working with. Hence, we require:

- A sort `Language`, that will be used by all the languages.
- Constants of sort `Language` for each language. In our case we have defined, in the module in charge of handling the database (`THM-DATABASE-HANDLING`) the following:

```
sort Language .
ops maude cafeobj : -> Language [ctor] .
```

- An attribute to store the selected language. We have added, also in `THM-DATABASE-HANDLING`, the following attribute:

```
op language`:_ : Language -> Attribute [ctor] .
```

- Commands to deal with this attribute. In this case, we have to add two new commands into the syntax (in `PROVE-COMMANDS`):

```
op maude`language`. : -> @Command@ .
op cafeOBJ`language`. : -> @Command@ .
```

- Rules for dealing with these commands. In our case we require the following two rules to be added to `THM-DATABASE-HANDLING`:

```
rl [maude-specs] :
   < O : X@Database | input : ('maude`language`..@Command@), output : nil ,
                      language : L, Atts >
=> < O : X@Database | input : nilTermList, output : ('\n '\b 'Maude 'selected
                                                     'as 'current 'specification
                                                     'language. '\o '\n),
                      language : maude, Atts > .

rl [cafe-specs] :
   < O : X@Database | input : ('cafeOBJ`language`..@Command@), output : nil ,
                      language : L, Atts >
=> < O : X@Database | input : nilTermList, output : ('\n '\b 'CafeOBJ 'selected
                                                     'as 'current 'specification
                                                     'language. '\o '\n),
                      language : cafeobj, Atts > .
```

- Every rule in charge of commands must use the `maude` value. Note that the rules in Section 2.1 used the `cafeobj` value.
- The attributes must be initialized when creating the initial system with a default value (in general the language must be `maude`, since it is the original language). We can just add the constant `initCafeAttS` defined in the previous section to the initial state:

```
rl [init] : init
=> [nil, < o : CITPDatabase | db : initialDatabase, input : nilTermList,
                              output : nil, default : 'CONVERSION,
                              pTree : null, currentGoal : 'nil,
                              showMod : false, tactic : 0,
                              tacticRec : ('SI 'CA 'CS 'TC 'IP),
                              language : maude, initCafeAttS >,
   ('\s '\s '\s '\s '\s '\b string2qidList(thm-banner) '\o '\n help-list)] .
```

Notice that these indications are quite general and most of them are only required the first time a new language is integrated.

Second, we have to merge our syntax with the tool syntax. This is achieved by:

- Adding the new syntax if any command is modified. In our case the `goal` command has extra syntax, since it supports transitions, so we add:

```
op trans_=>_; : @Bubble@ @Bubble@ -> @SentenceSet@ .
op ctrans_=>_if_; : @Bubble@ @Bubble@ @Bubble@ -> @SentenceSet@ .
op trns_=>_; : @Bubble@ @Bubble@ -> @SentenceSet@ .
op ctrns_=>_if_; : @Bubble@ @Bubble@ @Bubble@ -> @SentenceSet@ .
```

- Merging the modules containing the syntax for each tool. Since the syntax is stored into the metarepresented module `thm-Grammar`, we can use the following equation:

```
eq thm-Grammar = addImports((including 'PROVE-COMMANDS .), CafeGRAMMAR) .
```

  In this way `thm-Grammar` contains Full Maude syntax (that was already contained in `CafeGRAMMAR`), the syntax to support CafeOBJ specifications and the syntax for the CITP.

Third, we have to combine the behavior of both modules, so all the rules can be applied. Since the rules dealing with CITP commands are executed by an object of class `CITPDatabaseClass`, a subsort of `DatabaseClass`, we can add a new subsort to merge both databases:

```
subsort CITPDatabaseClass < CafeDatabaseClass .
```

After applying this modifications, the languages are integrated and both of them can be used independently.

References

[1] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *All About Maude: A High-Performance Logical Framework*, volume 4350 of *Lecture Notes in Computer Science*. Springer, 2007.

[2] K. Futatsugi and R. Diaconescu. *CafeOBJ Report*. World Scientific, AMAST Series, 1998.

[3] A. Riesco. *Introducing CafeOBJ specifications in the Full Maude database. Integrating CafeOBJ with the Constructor-based Inductive Theorem Prover—User Guide*. Universidad Complutense de Madrid, August 2013.