Using CafeOBJ specifications in Maude

Adrián Riesco Rodríguez
ariesco@fdi.ucm.es

Dept. Sistemas Informáticos y Computación
Universidad Complutense de Madrid, Spain

Japan Advanced Institute for Science and Technology
September 2013

Motivation

- CafeOBJ and Maude are sister languages of OBJ language.
- They are formal specification languages for system verification.
- Properties on CafeOBJ specifications can be verified by using proof scores.
- Maude provides other tools for debugging, theorem proving, real-time model checking, etc.
- Most of these tools are implemented on top of Full Maude, an extension of Maude implemented in Maude itself.
- We present here a general way to connect CafeOBJ specifications with these Maude tools.

CafeOBJ

- CafeOBJ is a language for writing formal specifications and verifying properties of them.
- It implements equational logic by rewriting.
- Different modules can be specified depending on the semantics we want to define:
  - `mod*` describe specifications with loose semantics.
  - `mod!` describe specifications with tight semantics.
- Modules can also be parameterized by using a module with loose semantics.

CafeOBJ

- Views are used to instantiate parameterized modules.
- CafeOBJ can be used as a powerful interactive theorem proving system.
- Specifiers can write proof scores also in CafeOBJ and doing proofs by executing the proof scores.
- It provides features such as mix-fix syntax, a typing system with ordered sorts, on-the-fly variables, syntactic sugar for predicates, etc.

CafeOBJ

- We describe CafeOBJ specifications by means of an example.
- We specify first sets of natural numbers.
- CafeOBJ specification start with a keyword indicating the kind of module followed by the module identifier:

```
mod! NSET {
```

- Other modules can be imported with the modes `protecting`, `extending`, `including`, and `using`:

```
pr(NAT)
```

- Sorts and subsorts are defined by using square brackets:

```
[Nat < Set]
```

CafeOBJ

- We define terms of these sorts by means of operators with the `constr` attribute.
- They can also have equational axioms such as being commutative `comm` or associative `assoc`:

```
op empty : -> Set {constr}
op _,_ : Set Set -> Set {constr comm assoc id: (empty)}
```

- We can also define functions over these sorts.
- The equations defining their behavior can use variables defined as follows:

```
var N : Nat
var S : Set
```

CafeOBJ

- The function |_| computes the size of a set:

```
op |_| : Set -> Nat
eq [s1] : | empty | = 0 .
eq [s2] : | N, S | = s(| S |) .
```

- Analogously, the predicate empty? checks whether a set is empty:

```
pred empty? : Set
eq [e1] : empty?(empty) = true .
eq [e2] : empty?((N, S)) = false .
}
```

CafeOBJ

- We can use the set to define a simple module, called `BLACKBOARD`.
- It just simulates a blackboard where numbers can be replaced by others.

```
mod! BLACKBOARD {
 pr(NSET)

 vars N N' : Nat
 var  S : Set
```

- The dynamic behavior of the system is described by means of transitions.
- The rule add takes any two numbers in the set (using the `comm` attribute from the constructor), adds them and divides the result by two:

```
 trans [add] : N, N', S => ((N + N') quo 2), S .
}
```

The Maude system

- Maude is a high-level language and high-performance system supporting both equational and rewriting logic computation.
- Maude modules correspond to specifications in rewriting logic.
- This logic is an extension of membership equational logic (MEL).
  - Sorts are grouped into equivalence classes called *kinds*.
  - We have equations, that must be confluent and terminating.
  - We can state membership axioms characterizing the elements of a sort.
  - Maude functional modules correspond to specifications in MEL.
- Rewriting logic extends MEL by adding rewrite rules.
  - Rules have to be coherent with equations.
  - Maude system modules correspond to specifications in rewriting logic.

The Maude system

- As shown for CafeOBJ, in Maude we distinguish between:
  - Theories, with loose semantics.
  - Modules, with tight semantics.
- Modules (but not theories) can be parameterized.
- Views are used to instantiate parameterized modules.

The Maude system

- The `NSET` module is defined in Maude as a functional module, with syntax `fmod`:

```
fmod NSET is
  pr NAT .
```

- Sorts and subsorts are declared with the keywords `sorts` and `subsort`:

```
sort Set .
subsort Nat < Set .
```

- Operators are declared with CafeOBJ-like syntax:

```
op empty : -> Set [ctor] .
op _,_ : Set Set -> Set [ctor comm assoc id: empty] .
```

The Maude system

- Variables and equations are also similar to ones previously shown:

```
  var N : Nat .
  var S : Set .

  op |_| : Set -> Nat .
  eq [s1] : | empty | = 0 .
  eq [s2] : | N, S | = s(| S |) .
```

- While predicates are just Boolean fuctions:

```
  op empty? : Set -> Bool .
  eq [e1] : empty?(empty) = true .
  eq [e2] : empty?((N, S)) = false .
endfm
```

The Maude system

- The `BLACKBOARD` module is now defined in a system module, using the keyword `mod`, which indicates that it contains rules:

```
mod BLACKBOARD is
  pr NSET .
```

- Rules are defined as follows:

```
  vars N N' : Nat .
  var  S : Set .

  rl [add] : N, N', S => ((N + N') quo 2), S .
endm
```

The Maude system

- We can parse terms with `parse`:

```
Maude> parse 3, 4, 5 .
Set: 3,4,5
```

- Equations are executed with the `red` command:

```
Maude> red | 3, 5, 6, 7, 8 | .
result NzNat: 5
```

- Rules are applied with `rew`:

```
Maude> rew 3, 5, 6, 7, 8 .
result NzNat: 7
```

The Maude system

- Searches using rules are performed with `search`:

```
Maude> search 3, 5, 6, 7, 8 =>* N:Nat s.t. N:Nat =/= 7 .
Solution 1 (state 46)
N --> 6
Solution 2 (state 47)
N --> 5
...
```

- We can also see the path followed to reach these results:

```
Maude> show path 46 .
state 0, Set: 3,5,6,7,8
===[ rl N,N',S => S,(N + N') quo 2 [label add] . ]===>
state 1, Set: 4,6,7,8
===[ rl N,N',S => S,(N + N') quo 2 [label add] . ]===>
state 11, Set: 5,7,8
===[ rl N,N',S => S,(N + N') quo 2 [label add] . ]===>
state 32, Set: 6,7
===[ rl N,N',S => S,(N + N') quo 2 [label add] . ]===>
state 46, NzNat: 6
```

CafeOBJ vs. Maude

```
mod! NSET {
 pr(NAT)
 [Nat < Set]

 op empty : -> Set {constr}
 op _,_ : Set Set -> Set
   {constr comm assoc id: (empty)}

 var N : Nat
 var S : Set

 op |_| : Set -> Nat
 eq [s1] : | empty | = 0 .
 eq [s2] : | N, S | = s(| S |) .

 pred empty? : Set
 eq [e1] : empty?(empty) = true .
 eq [e2] : empty?((N,S)) = false .
}
```

```
fmod NSET is
 pr NAT .
 sort Set .  subsort Nat < Set .

 op empty : -> Set [ctor] .
 op _,_ : Set Set -> Set
   [ctor comm assoc id: empty] .

 var N : Nat .
 var S : Set .

 op |_| : Set -> Nat .
 eq [s1] : | empty | = 0 .
 eq [s2] : | N, S | = s(| S |) .

 op empty? : Set -> Bool .
 eq [e1] : empty?(empty) = true .
 eq [e2] : empty?((N,S)) = false .
endfm
```

CafeOBJ vs. Maude

```
mod! BLACKBOARD {
 pr(NSET)

 vars N N' : Nat
 var  S : Set

 trans [add] : N, N', S
 => ((N + N') quo 2), S .
}
```

```
mod BLACKBOARD is
 pr NSET .

 vars N N' : Nat .
 var  S : Set .

 rl [add] : N, N', S
 => ((N + N') quo 2), S .
endm
```

CafeOBJ vs. Maude

- We can also find a number of differences:
  - Maude is built on top of Membership Equational Logic. Given this specification for ordered lists:

```
sorts List OList .
subsort Nat < OList < List .

op nil : -> OList [ctor] .
op __ : List List -> List [ctor assoc id: nil] .
```

  - We use membership axioms stating the sort of a term:

```
cmb N N' L : OList
 if N <= N' /\
    N' L : OList .
```

CafeOBJ vs. Maude

- CafeOBJ loose specifications can be parameterized, while Maude theories cannot.
- CafeOBJ importation system is more flexible, it allows to import other modules into loose specifications in any mode, while Maude only allows `including`.
- Maude allows transitions to be used as conditions for other transitions, instantiating the new variables appearing in the righthand side of this condition.
- Maude provides attributes such as `otherwise`, which indicates that the equation is only applied if all the others fail, and `metadata`, which allows provides extra information for the programmed when working at the metalevel.

Reflection

- Rewriting logic is reflective, because there is a finitely presented rewrite theory $\mathcal{U}$ that is universal in the sense that:
  - we can represent any finitely presented rewrite theory $\mathcal{R}$ and any terms t, t' in $\mathcal{R}$ as terms $\overline{\mathcal{R}}$ and $\overline{t}, \overline{t'}$ in $\mathcal{U}$,
  - then we have the following equivalence

$$\mathcal{R} \vdash t \longrightarrow t' \iff \mathcal{U} \vdash \langle \overline{\mathcal{R}}, \overline{t} \rangle \longrightarrow \langle \overline{\mathcal{R}}, \overline{t'} \rangle.$$

- Since $\mathcal{U}$ is representable in itself, we get a reflective tower

$$\begin{array}{c} \mathcal{R} \vdash t \to t' \\ \Updownarrow \\ \mathcal{U} \vdash \langle \overline{\mathcal{R}}, \overline{t} \rangle \to \langle \overline{\mathcal{R}}, \overline{t'} \rangle \\ \Updownarrow \\ \mathcal{U} \vdash \langle \overline{\mathcal{U}}, \overline{\langle \overline{\mathcal{R}}, \overline{t} \rangle} \rangle \to \langle \overline{\mathcal{U}}, \overline{\langle \overline{\mathcal{R}}, \overline{t'} \rangle} \rangle \\ \vdots \end{array}$$

Maude's metalevel

In Maude, key functionality of the universal theory $\mathcal{U}$ has been efficiently implemented in the functional module `META-LEVEL`:

- Maude terms are reified as elements of a data type `Term` in the module `META-TERM`;
- Maude modules are reified as terms in a data type `Module` in the module `META-MODULE`;
- operations `upModule`, `upTerm`, `downTerm`, and others allow moving between reflection levels;

Maude's metalevel

- the process of reducing a term to canonical form using Maude's `reduce` command is metarepresented by a built-in function `metaReduce`;
- the processes of rewriting a term in a system module using Maude's `rewrite` and `frewrite` commands are metarepresented by built-in functions `metaRewrite` and `metaFrewrite`;
- the process of applying a rule of a system module at the top of a term is metarepresented by a built-in function `metaApply`;

Maude's metalevel

- the process of applying a rule of a system module at any position of a term is metarepresented by a built-in function `metaXapply`;
- the process of matching two terms is reified by built-in functions `metaMatch` and `metaXmatch`;
- the process of searching for a term satisfying some conditions starting in an initial term is reified by built-in functions `metaSearch` and `metaSearchPath`; and
- parsing and pretty-printing of a term in a module, as well as key sort operations such as comparing sorts in the subsort ordering of a signature, are also metarepresented by corresponding built-in functions.

Metalevel syntax

- Sorts and kinds are metarepresented as data in specific subsorts of the sort `Qid` of quoted identifiers (i.e., terms starting with the `'` character, like `'Nat`).
- Terms can be either:
  - Constants, which are quoted identifiers containing the constants name and its type separated by a `.` (e.g., `'0.Nat`).
  - Variables, which contain their name and type separated by a `:` (e.g., `'X:Nat`).
  - A built term is constructed by applying an operator symbol (a `Qid`) to a list of terms (e.g., `'_+_['0.Zero, 'N:Nat]`):

```
sort TermList .
subsort Term < TermList .
op _,_ : TermList TermList -> TermList
    [ctor assoc gather (e E) prec 120] .
op _[_] : Qid TermList -> Term [ctor] .
```

Metalevel syntax

- For metarepresenting modules, we have the sort `Module`:

```
op fmod_is_sorts_.____endfm : Header ImportList SortSet
      SubsortDeclSet OpDeclSet MembAxSet EquationSet -> Module
      [ctor gather (& & & & & & &)] .
```

- The constructors for the sorts used above follow Maude syntax:

```
op eq_=_[_]. : Term Term AttrSet -> Equation [ctor] .
op ceq_=_if_[_]. : Term Term EqCondition AttrSet -> Equation
      [ctor] .
```

- For example, we metarepresent `eq [add1] 0 + N = N .` as a term of sort `Equation`:

```
eq '_+_['0.Zero, 'N:Nat] = 'N:Nat [label('add1)] .
```

Representing modules: Example at the object level

```
fmod NSET is
 pr NAT .
 sort Set .
 subsort Nat < Set .

 op empty : -> Set [ctor] .
 op _,_ : Set Set -> Set [ctor comm assoc id: empty] .

 var N : Nat .
 var S : Set .

 op |_| : Set -> Nat .
 eq [s1] : | empty | = 0 .
 eq [s2] : | N, S | = s(| S |) .

 op empty? : Set -> Bool .
 eq [e1] : empty?(empty) = true .
 eq [e2] : empty?((N, S)) = false .
endfm
```

Representing modules: Example at the metalevel

```
fmod 'NSET is
  including 'BOOL .
  protecting 'NAT .
  sorts 'Set .
  subsort 'Nat < 'Set .

  op '_`,_ : 'Set 'Set -> 'Set [assoc comm ctor id('empty.Set)] .
  op 'empty : nil -> 'Set [ctor] .
  op 'empty? : 'Set -> 'Bool [none] .
  op '|_| : 'Set -> 'Nat [none] .

  none

  eq 'empty?['empty.Set] = 'true.Bool [label('e1)] .
  eq 'empty?['_`,_['N:Nat,'S:Set]] = 'false.Bool [label('e2)] .
  eq '|_|['empty.Set] = '0.Zero [label('s1)] .
  eq '|_|['_`,_['N:Nat,'S:Set]] = 's_['|_|['S:Set]] [label('s2)] .
endfm
```

Representing modules: Example at the object level

```
mod BLACKBOARD is
 pr NSET .

 vars N N' : Nat .
 var  S : Set .

 rl [add] : N, N', S => ((N + N') quo 2), S .
endm
```

Representing modules: Example at the metalevel

```
mod 'BLACKBOARD is
  including 'BOOL .
  protecting 'NSET .
  sorts none .
  none
  none
  none
  none
  rl '_`,_['N:Nat,'N':Nat,'S:Set] =>
     '_`,_['S:Set,'_quo_['_+_['N:Nat,'N':Nat],'s_^2['0.Zero]]]
     [label('add)] .
endm
```

Moving between levels

```
op upModule : Qid Bool ~> Module [special (...)] .
op upSorts : Qid Bool ~> SortSet [special (...)] .
op upSubsortDecls : Qid Bool ~> SubsortDeclSet [special (...)] .
op upOpDecls : Qid Bool ~> OpDeclSet [special (...)] .
op upMbs : Qid Bool ~> MembAxSet [special (...)] .
op upEqs : Qid Bool ~> EquationSet [special (...)] .
op upRls : Qid Bool ~> RuleSet [special (...)] .
```

In all these (partial) operations

- The first argument is expected to be a module name.
- The second argument is a Boolean, indicating whether we are interested also in the imported modules or not.

Moving between levels: Terms

```
fmod UP-DOWN-TEST is protecting META-LEVEL .
  sort Foo .
  ops a b c d : -> Foo .
  op f : Foo Foo -> Foo .
  op error : -> [Foo] .
  eq c = d .
endfm

Maude> reduce in UP-DOWN-TEST : upTerm(f(a, f(b, c))) .
result GroundTerm: 'f['a.Foo,'f['b.Foo,'d.Foo]]

Maude> reduce downTerm('f['a.Foo,'f['b.Foo,'c.Foo]], error) .
result Foo: f(a, f(b, c))

Maude> reduce downTerm('f['a.Foo,'f['b.Foo,'e.Foo]], error) .
Advisory: could not find a constant e of
          sort Foo in meta-module UP-DOWN-TEST.
result [Foo]: error
```

metaParse

- Its first argument is the representation in META-LEVEL of a module $\mathcal{R}$, its second argument is a list of quoted identifiers, and the third one is a type.
- It tries to parse the given list of tokens as a term of the given type in the module given as first argument.
- The constant anyType allows any component.
- If metaParse succeeds, it returns the metarepresentation of the parsed term.
- Otherwise, it returns, noParse or ambiguity.

```
Maude> reduce in META-LEVEL : metaParse(upModule('BLACKBOARD, false),
                                        '3 '`, '4 '`, '5, anyType) .

result ResultPair: {'_`,_['s_^3['0.Zero],'_`,_['s_^4['0.Zero],
                           's_^5['0.Zero]]],'Set}
```

metaReduce

- Its first argument is the representation in META-LEVEL of a module $\mathcal{R}$ and its second argument is the representation in META-LEVEL of a term t.
- It returns the metarepresentation of the normal form of t, using the equations in $\mathcal{R}$, together with the metarepresentation of its corresponding sort or kind.
- The reduction strategy used by metaReduce coincides with that of the reduce command:

```
Maude> reduce in META-LEVEL : metaReduce(upModule('NSET, false),
            '|_|['_',_['0.Zero,'s_['0.Zero],'s_['s_['0.Zero]]]]) .

result ResultPair: {'s_^3['0.Zero],'NzNat}
```

metaRewrite

- Its first two arguments are the representations in META-LEVEL of a module $\mathcal{R}$ and of a term t, and its third argument is a natural n.
- Its result is the representation of the term obtained from t after at most n applications of the rules in $\mathcal{R}$ using the strategy of Maude's command rewrite, together with the metarepresentation of its corresponding sort or kind.

```
Maude> reduce in META-LEVEL :
         metaRewrite(upModule('BLACKBOARD, false),
           '_`,_['s_^2['0.Zero], '_`,_['s_^3['0.Zero],
           '_`,_['s_['0.Zero], '0.Zero]]], 1) .
result ResultPair:
  {'_`,_['0.Zero,'s_^2['0.Zero],'s_^3['0.Zero]], 'Set}

Maude> reduce in META-LEVEL :
         metaRewrite(upModule('BLACKBOARD, false),
           '_`,_['s_^2['0.Zero], '_`,_['s_^3['0.Zero],
           '_`,_['s_['0.Zero], '0.Zero]]], 5) .
result ResultPair:
  {'s_^2['0.Zero], 'NzNat}
```

The Maude system: Full Maude

- Full Maude is an extension of Maude implemented in Maude itself.
- It provides a more powerful and extensible module algebra than that available in Core Maude.
- It includes object-oriented modules (which can be parameterized) with support notation for objects, messages, classes, and inheritance.
- Full Maude has an explicit database for modules, which allows the user to manipulate them.
- It uses the Loop Mode module, generating an input/output loop to interact with the user.
- The Loop Mode forces us to introduce commands and modules by enclosing them in parentheses.
- All the commands available in Core Maude are available in Full Maude, and executed at the metalevel.

The Maude system: Full Maude

- Full Maude itself can be used as a basis for further extensions, by adding new functionality.
- It is possible both to change the syntax or the behavior of existing features, and to add new features.
- For example, we can add new modules and new commands.
- In this way Full Maude becomes a common infrastructure on top of which one can build tools.
- Examples of new applications are the Maude Declarative Debugger, the Real-Time Maude tool, and the Constructor-based Inductive Theorem Prover.

Objectives

- We present here an extension of Full Maude to accept CafeOBJ specifications.
- We have defined the syntax of CafeOBJ modules, so it can be parsed.
- These modules are then translated (when possible) into Maude modules.
- We have also created new commands to work with the original CafeOBJ modules.
- We use this as basis for extending other tools.

Translation

- We translate:
  - Modules with loose semantics as theories.
  - Modules with tight semantics as modules.
  - Importantions, sorts, subsorts, operators (and their attributes), and equations as themselves.
  - Predicates as Boolean operators.
  - Transitions as rules.
  - Since Maude does not support parameterized theories, we translate this kind of module as a parameterized module, showing a message.
  - Similarly, we change the importation mode for theories to `including`, showing a message warning the user.

CafeOBJ: Syntax

- The first thing we need to do is defining CafeOBJ syntax.
- We have to declare the sorts the constructors:

```
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@ .
```

CafeOBJ: Syntax

- More specifically, we have sorts for:
  - Modules: `@CafeMODULE@`.
  - Sorts: `@VisibleSortDecl@`.
  - Variable declarations: `@CafeVarDecl@`.
  - Equations: `@CafeEqDecl@`.
  - Etc.
- We have also define all the subsort relations:
  - For types: `subsort @CafeToken@ < @CafeSort@ < @CafeType@ .`
  - For view identifiers: `subsort @CafeModExp@ < @CafeViewId@ .`
  - For indicating the modules are the data type to be introduced: `subsort @CafeMODULE@ < @Input@ .`
  - Etc.

CafeOBJ: Syntax

- The next step is defining the constructors.
- These constructors just follow the CafeOBJ syntax from the manual (taking into account the latest changes).
- For example, modules are defined as follows:
  - For loose semantics:

```
op mod*_{_} : @CafeInterface@ @CafeDeclList@
              -> @CafeMODULE@ [ctor] .
```

  - For tight semantics

```
op mod!_{_} : @CafeInterface@ @CafeDeclList@
              -> @CafeMODULE@ [ctor] .
```

  - For views:

```
op view_from_to_{_} : @CafeToken@ @CafeModExp@ @CafeModExp@
                      @CafeViewDeclList@ -> @CafeMODULE@ [ctor] .
```

- Shortcuts for creating modules are translated into terms with this syntax.

CafeOBJ: Syntax

- Importation modes are declared as follows:

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

- Equations have the following syntax:

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

CafeOBJ: Syntax

- Note the sort `@CafeBubble@` used for the lefthand side, the righthand side, and the condition.
- It indicates that they can be anything.
- It is used because these terms do not depend on the CafeOBJ syntax, but on the specific syntax defined in the module.
- We have to parse these bubbles later to:
  - Check that the terms are defined with valid functions.
  - Look for labels and attributes.
  - Check that the equation is executable (no new variables are used in the righthand side).

Parsing

- When the user introduces a CafeOBJ modules, it is into Full Maude as a list of `Qid`.
- For example, the module

```
mod! TALK {
  [Foo]

  op a : -> Foo {constr}
  var F : Foo

  op f : Foo -> Foo
  eq f(F) = a .
}
```

is read as:

```
'mod! 'TALK '`{ '`[ 'Foo '`] 'op 'a ': '-> 'Foo '`{ 'constr '`}
'var 'F ': 'foo
'op 'f ': 'Foo '-> 'Foo 'eq 'f '`( 'F '`) '= 'a '. '`}
```

Parsing

- Using the CafeOBJ syntax to (meta)parse it, we obtain

```
'cmod!_‘{_‘}['CafeToken[''TALK.Qid],
'__['‘[_‘]['CafeToken[''Foo.Qid]],
'__['op_:‘->_‘{_‘}.['CafeToken[''a.Qid],'CafeToken[''Foo.Qid],
    'constr.@CafeAttr@],
'__['var_:_.['neCafeTokenList[''F.Qid],'CafeToken[''Foo.Qid]],
'__['op_:_->_.['CafeToken[''f.Qid],'CafeToken[''Foo.Qid],
               'CafeToken[''Foo.Qid]],
'eq_=_.['CafeBubble['__[''f.Qid,''‘(.Qid,''F.Qid,''‘).Qid]],
               'CafeBubble[''a.Qid]]]]]]
```

Parsing

- It is easy to translate the tokens.
- To translate the bubbles, we first create an intermediate module with the importations, the variables, and the operator declarations in the module.
- In the CafeOBJ case, we have to extend the parsing with the variables defined in the lefthand side of the equation/transition.
- Then, we use this module to solve the bubbles.
- In this case, we obtain `'f['F:Foo]` from `'CafeBubble['__['']f.Qid,''`(.Qid,''F.Qid,''`).Qid]]`.
- We have to check later that the executability restrictions hold.

Parsing

- For example, predicates are parsed as Maude operators:

```
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) .
```

where `map2MaudeAttr` translates the CafeOBJ attributes:

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

Parsing

- Transitions are translated as rules as follows:

```
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') .
```

where `opDeclSetFromQidList` extends the set of variables with the ones declared on-the-fly on the lefthand side.

After parsing

- Once we can introduce CafeOBJ modules into the database, we can use any Maude command with them.
- Moreover, we can define our own commands to interact with these modules.
- For example, it might be interesting to define a command to show the original module, since the Maude commands show the translated one.
- This is done by:
  - Extending the Full Maude syntax for commands.
  - Defining rewrite rules dealing with this new commands.
  - In this case we also need to define a new attribute to keep the original module, since some translations cannot be undone (e.g. we do not know which Boolean operators were originally predicates).

Accepted modules

- For some CafeOBJ modules the translation might not be accurate from the semantics point of view.
- We use the syntax of the latest CafeOBJ release, so matching conditions can be used in equations and transitions.
- We also accept statements using the `nonexec` attribute.
- A previous step of pre-processing allows the user to introduce `metadata` information.

Integration

EXAMPLE

Integration with other tools

- We can now proceed to integrate this extension with other tools built on top of Full Maude.
- The easiest way to do it is just use with these tools the Maude modules obtained after the translation.
- This is achieved by:
  - Extending the syntax of the tool with the one for CafeOBJ.
  - Adding the rules for dealing with these specification by using a subsort of the `CafeHandler`.

Integration with other tools

- This simple mechanism allows us to test and debug CafeOBJ specifications with Maude tools.
- The test-case generator builds ground terms for the given function.
- The results can be checked by the user or against a correct specification.
- Declarative debugging builds a tree corresponding to an erroneous computation.
- It asks questions about the subcomputations (stored in the nodes of the tree) until the error is found.

Integration with other tools

- Given the following specification:

```
mod! LIST {
  pr(NAT)
  [Nat < List]

  op nil : -> List {constr}
  op __ : List List -> List {constr assoc id: (nil)}

  var N : Nat
  var L : List

  op reverse : List -> List
  eq [r1] : reverse(nil) = 0 .
  eq [r2] : reverse(N L) = reverse(L) N .
}
```

Integration with other tools

- We can first generate test cases by selecting a coverage criterium.

```
Maude> (function coverage .)
Function Coverage selected
```

  which indicates that all the calls to a given function must use all the possible equations defining it.

- Then, we can test the `reverse` function as follows:

```
Maude> (test reverse .)

1 test cases have to be checked by the user:
    1. The term reverse(0 0) has been reduced to 0 0 0

All calls were covered.
```

Integration with other tools

- Since it is not an expected result, we can debug it with the command:

```
Maude> (debug reverse(0) -> 0 0 .)
```

- Maude builds the following tree, where the application of each equation has as children the equations directly applied when executing it:

$$\dfrac{\dfrac{\dfrac{}{\texttt{reverse(nil)} \rightarrow \texttt{0}}\ \text{r1}}{\texttt{reverse(0)} \rightarrow \texttt{0 0}}\ \text{r2}}{\texttt{reverse(0 0)} \rightarrow \texttt{0 0 0}}\ \text{r2}$$

Integration with other tools

- The tool asks the following questions:

```
Is this reduction (associated with the equation r2) correct?

reverse(0) -> 0 0

Maude> (no .)

Is this reduction (associated with the equation r1) correct?

reverse(nil) -> 0

Maude> (no .)
```

- With these answers it finds the error:

```
The buggy node is:
reverse(nil) -> 0
with the associated equation: r1
```

Integration

EXAMPLE

Integration with other tools

- We can also integrate other tools but rebuilt the interface to hide Maude and use CafeOBJ syntax.
- It requires, in addition to the changes above, to add a new attribute distinguishing the language we are using.
- Then, we have to redefine all (or most of) the commands provided by the tool to use our syntax.

Integration with other tools

- For example, we have the Constructor-based Inductive Theorem Prover (CITP).
- It can prove properties on Maude specifications.
- These properties are described by means of equations, membership axioms, or rules in Maude syntax.
- However, in our case we are interested on equations and transitions in CafeOBJ syntax.
- Hence, we have to define rules in charge of processing this command.
- It is also necessary to define rules for errors.

Integration with other tools

The rule for introducing goals in CafeOBJ is defined as follows:

```
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! .
```

Integration with other tools

- The `input` is the same as the one for Maude specifications.
- The flag for `language` is `cafeobj`.
- We use the function `isCafeMod` to check whether it is a valid CafeOBJ.
- The command is processed with a new function `procGoalsCafe`, which generates the goal updates the database.
- The goal is printed with the function `prettyPrintProofTreeCafe`, which generates CafeOBJ-like output.
- For this purpose we use the original CafeOBJ module, stored in the `originalCafeModule` attribute.

Integration with other tools

- Given a theory stating the existence of a sort:

```
mod* ELT {
 [Elt]
}
```

- And a simple module specifying lists:

```
mod! LISTS(X :: ELT) {
  [List]
  op nil : -> List {constr}
  op __ : Elt.X List -> List {constr}
```

- With composition of lists:

```
  op _@_ : List List -> List
  eq nil @ L2 = L2 .
  eq (E L1) @ L2 = E (L1 @ L2) . *** [metadata "CA-1"] .
```

Integration with other tools

- And a non-deterministic function that combines two different lists:

```
  op mix : List List -> List {comm}

  trans [m1] : mix(nil, L) => L .
  trans [m2] : mix(E L1, L2) => E mix(L1, L2) .
}
```

- We can indicate that we are working with CafeOBJ specifications:

```
Maude> (cafeOBJ language .)
CafeOBJ selected as current specification language.
```

Integration with other tools

- And define goals with transitions:

```
Maude> (goal LISTS |- trans mix(L1:List, nil) => L1 ;)
====================== GOAL 1-1 ======================
< Module LISTS is concealed
...
end
,
  trans mix(L1:List,nil)
    => L1:List . >
unproved

INFO: an initial goal generated!
```

Integration with other tools

- Then we can use the standard commands to prove it:

```
Maude> (auto .)

INFO: Goal 1-1 was successfully proved
by applying tactic: SI CA CS TC IP

INFO: PROOF COMPLETED
```

Integration

EXAMPLE

Concluding remarks

- We have implemented an extension of Full Maude to parse CafeOBJ specifications.
- The standard Maude commands, such as `reduce`, `rewrite`, `search`, and `modelCheck` can be used with these specifications.
- This extension eases the integration of CafeOBJ specification with any tool implemented in top of Full Maude.
- We have shown how to integrate tools from the implementation point of view, but we have to devote some time to check that we are using these tools appropriately from the theoretical point of view.