CafeOBJ Reference Manual

Toshimi Sawada, Kokichi Futatsugi, Norbert Preining

2014-02-05

Contents

Chapter 1

Introduction

This manual introduces the language CafeOBJ. Is is a reference manual with the aim to document the current status of the language, and not targetting at an exhaustive presentation of the mathematical and logical background. Still, the next section will give a short summary of the underlying formal approach and carry references for those in search for details.

The manual is structured into three parts. The first one being this introduction, the second one being the presenation of basic concepts of CafeOBJ by providing a simple protocol which will get specified and verified. Although the second part tries to give a view onto the core features and their usage, it should not be considered a course in CafeOBJ, and cannot replace a proper introduction to the language.

Finally, the last part consists of explanations of all current language elements in alphabetic order. This includes several higher level concepts, as well as heavy cross-referencing.

While we hope that this manual and the introductory part helps beginners to start programming in CafeOBJ, the main target are those who already have acquired a certain level of fluency, but are in need for a reference of the language.

Background of CafeOBJ

CafeOBJ is a specification language based on three-way extensions to many-sorted equational logic: the underlying logic is order-sorted, not just many-sorted; it admits unidiretional transitions, as well as equations; it also accommodates hidden sorts, on top of ordinary, visible sorts. A subset of CafeOBJ is executable, where the operational semantics is given by a conditional order-sorted term rewriting system. These theoretical bases are indispensable to employ CafeOBJ properly. Fortunately, there is an ample literature on these subjects, and we are able to refer the reader to, e.g., [4], [12] (for basics of algebraic specifications), [8], [6] (for order-sorted logic), [7] (for hidden sorts), [9] (for coinduction), [11] (for rewriting logic), [5] (for institutions), and [10], [1] (for term rewriting systems), as primers. The logical aspects of CafeOBJ are explained in detail in [3] and [2]. This manual is for the initiated, and we sometimes abandon the theoretical rigour for the sake of intuitiveness.

For a very brief introduction, we just highlight a couple of features of CafeOBJ. CafeOBJ is an offspring of the family of algebraic specification techniques. A specification is a text, usually of formal syntax. It denotes an algebraic system constructed out of sorts (or data types) and sorted (or typed) operators. The system is characterised by the axioms in the specification. An axiom was traditionally a plain equation ("essentially algebraic"), but is now construed much more broadly. For example, CafeOBJ accommodates conditional equations, directed transitions, and (limited) use of disequality.

The underlying logic of CafeOBJ is as follows:

Order-sorted logic [8] A sort may be a subset of another sort. For example, natural numbers may be embedded into rationals. This embedding makes valid the assertion that 3 equals 6/2. It also realises "operator inheritance", in the sense that an operator declared on rationals are automatically declared on natural numbers. Moreover, the subsort relation offers you a simple way to define partial operations and exception handling.

Rewriting logic [11] In addition to equality, which is subject to the law of symmetry, you may use transition relations, which are directed in one way only. State transitions are naturally formalised by those relations. In particular, transition relations are useful to represent concurrency and/or indeterminacy.

Hidden sorts [7] You have two kinds of equivalence. One is a minimal equivalence, that identifies terms (elements) iff they are the same under the given equational theory. Another equivalence, employed for so-called hidden sorts, is behavioural: two terms are equivalent iff they behave identically under the given set of observations.

We would also like to emphasise a very useful feature of CafeOBJ.

Parameters There are many sorts that are inherently generic. Stacks, lists, sets and so on have operations that act independently of the properties of base ("data") elements. A more tricky case is priority queues, which require base elements to have an order relation. You may define these sorts by parameterised modules, where base elements are parameterised out. A parameter may be subject to constraints. For example, the parameter of a priority queue module may be declared an ordered set, not an arbitrary set.

Chapter 2

Overview of the system

Let us start with a simple definition of a module, which are the basic building blocks of any CafeOBJ program:

```
mod NATPAIR {
  pr(NAT)
  [Pair]
  var P : Pair
  op <_,_> : Nat Nat -> Pair {constr}
  op fst : Pair -> Nat
  op snd : Pair -> Nat
  eq fst( < A:Nat , B:Nat > ) = A .
  eq snd( < A:Nat , B:Nat > ) = B .
}
```

This example already presents most of the core concepts of CafeOBJ:

- modules as the basic building blocks
- import of other modules `pr(NAT)`
- sorts `[Pair]`
- operator signature and equations

Let us start with sorts, as they are the fundamental types.

Sorts

Most programming languages allow for different sorts, or types of objects. In this respect CafeOBJ is not different and allows to have arbitrary sorts. In addition, these sorts can be ordered, more specific one sort can be declared a sub-sort of another. In the above example

```
[ Pair ]
```

a new sort called `Pair` is introduced. This is a completely new sort and is in no sub-sort relation to any other sort. This is a very common case, and reflects the different types of objects in other programming languages.

In case one wants to introduce ordering in the sorts, the order can be expressed together with the definition of the sort, as in:

```
[ Nat < Set ]
```

which would introduce a new sort `Set` and declares it as supersort of the (builtin) sort `Nat`.

For more details concerning sorts, see `sort declaration`.

Imports

CafeOBJ allows for importing and reusing of already defined modules:

```
pr(NAT)
```

for example pulls in the natural numbers (in a very minimal implementation). There are several modes of pulling in other modules, differing in the way the (semantic) models of the included module are treated.

After a statement of import, the sorts, variables, and operators of the imported modules can be used.

For more details see `protecting`, `extending`, `using`, `including`

Variables and Operators

While sorts define data types, variables hold objects of a specific type, and operators define functionality. For each variable its sort has to be declared, and for each operator the signature, i.e., the sorts of the input data and the sort of the output, has to be given.

```
var P : Pair
op fst : Pair -> Nat
```

This example declares a variable `P` of type pair, and an operator `fst` which maps the sort `Pair` to the sort `Nat`, or in other words, a function that maps pairs of natural numbers to natural numbers.

We have seen already a different way to specify operators, namely

```
op <_,_> : Nat Nat -> Pair {constr}
```

which introduces an infix operator. CafeOBJ is very flexible and allows to freely specify the syntax. In an operator declaration as the above, the underscores _ represent arguments to the operator. That also means that the number of underscores must match the number of sorts given before the ->. After the above declaration CafeOBJ will be able to parse terms like `< 3 , 4 >` and correctly type them as pair.

For further details, see `var`, `op`.

Equations (or Axioms)

Using sorts, variables, and operators we have specified the terms that we want to speak about. In the following equations, or sometimes called axioms, will equate different terms. Equating here is meant in the algebraic sense, but also in the term-rewriting sense, as equations form the basis of rewrite rules which provide CafeOBJ with the executable semantics:

```
eq fst( < A:Nat , B:Nat > ) = A .
eq snd( < A:Nat , B:Nat > ) = B .
```

As soon as an operator like `fst` has been declared, we can give equations. In this case we define `fst` of a pair to return the first element.

For further details see `eq`.

In the following chapter we will include the specification of a protocol with the full code, explaining some concepts on the way.

to be written

alternative title: Main concepts (?)

discuss the following topics in bit more details (parts of the current manual, stripped down)

- sorts [Ch 3]
- operators [Ch 4, Ch 7]
- module [Ch 2, Ch 8]

do not contain the syntactic definition in all the details, but explain these important items in more detail

Chapter 3

CloudSync

In the following we will model a very simple protocol for cloud syncronization of a set of PCs. The full code of the actual specification, as well as parts of the verification proof score will be included and discussed.

Protocoll

One cloud computer and arbitrary many PCs have one value each that they want to keep in sync. This value is a natural number, and higher values mean more recent (like SVN revision numbers).

The Cloud can be in two states, *idle* and *busy*, while the PCs can be on of the following three states: *idle*, *gotvalue*, *updated*. The Cloud as well as all PCs are initially in the *idle* state. When a PC connects to the cloud, three things happen:

1. the cloud changes into *busy* state
2. the PC reads the value of the cloud and saves it in a temporary location
3. the PC changes into *gotvalue* state

In the *gotvalue* state the PC compares his own value against the value it got from the cloud, and updates accordingly (changes either the cloud or the own value to the larger one). After this the PC changes into the *updated* state.

From the *update* state both the Cloud and the PC return into the *idle* state.

TODO include a graphic that shows this TODO

Specification

We will now go through the full specification with explanations of some of the points surfacing. We are starting with two modules that specify the possible states the cloud and the PCs can be in:

```
mod! CLLABEL {
  [ClLabelLt < ClLabel]
  ops idlecl busy : -> ClLabelLt {constr} .
  eq (L1:ClLabelLt = L2:ClLabelLt) = (L1 == L2) .
}
mod! PCLABEL {
  [PcLabelLt < PcLabel]
  ops idlepc gotvalue updated : -> PcLabelLt {constr} .
  eq (L1:PcLabelLt = L2:PcLabelLt) = (L1 == L2) .
}
```

Both modules define two new sorts each, the actual label, and literals for the labels. One can see that we declare the signatures of the literal labels with the `ops` keyword, which introduces several operators of the same signature at the same time.

The predicate == is the equivalence predicate defined via reduction. Thus, the two axioms given above state that two literals for labels are the same if they are syntactically the same, since they cannot be rewritten anymore.

Furthermore, note that we choose different names for the *idle* state of the PCs and the cloud, to have easy separation.

The next module introduces a parametrized pair module. Parametrizing modules is a very powerful construction, and common in object oriented programming languages. In principle we leave open what are the actual components of the pairs, and only specify the operational behaviour on a single pair.

In this and the next example of the multi-set, there are no additional requirements on the sorts that can be used to instantiate a pair (or multi-set). In a more general setting the argument after the double colon `::` refers to a sort, and an instantiation must be adequate for this sort (details require deeper understanding of homomorphism).

```
mod! PAIR(X :: TRIV,Y :: TRIV) {
  [Pair]
  op <_,_> : Elt.X Elt.Y -> Pair {constr}
  op fst : Pair -> Elt.X
  op snd : Pair -> Elt.Y
  eq fst(< A:Elt.X,B:Elt.Y >) = A .
  eq snd(< A:Elt.X,B:Elt.Y >) = B .
}
```

The next module is also parametrized, axiomatizing the concept of multi-set where a certain element can appear multiple times in the multi-set. We want to use this module to present another feature, namely the option to specify additional properties of some operators. In this case we are specifying that the constructor for sets is associative `assoc`, commutative `comm`, and has as identity the `empty` set.

While it is easily possible to add associativity and commutativity as axioms directly, this is not advisable, especially for commutativity. Assume adding the simple equation `eq A * B = B * A .`. This defines a rewrite rule from left to right. But since `A` and `B` are variables the can be instantiated with arbitrary subterms, and one would end up with an infinite rewriting.

```
mod MULTISET(X :: TRIV) {
  [ Elt.X < MultiSet ]
  op empty : -> MultiSet {constr} .
  -- associative and commutative set constructor with identity empty
  op (_ _) : MultiSet MultiSet -> MultiSet { constr assoc comm id: empty }
}
```

With all this set up we can defined the cloud state as a pair of a natural number, and a state. Here we see how a parametrized module is instantiated. The details of the renaming for the second element are a bit involved, but thinking about renaming of sorts and operators to match the ones given is the best idea.

Having this in mind we see that when we put the `CLLABEL` into the second part of the pair, we tell the system that it should use the `ClLabel` sort for the instantiation of the `Elt` sort, and not the `ClLabelLt` sort.

Furthermore, after the instantiation we rename the final outcome again. In this case we rename the `Pair` to `ClState`, and the operators to their cousins with extension in the name.

```
mod! CLSTATE {
  pr(PAIR(NAT, CLLABEL{sort Elt -> ClLabel})*
    {sort Pair -> ClState, op fst -> fst.clstate, op snd -> snd.clstate })
}
```

The PC state is now very similar, only that we have to have a triple (`3TUPLE` is a builtin predicate of CafeOBJ), since we need one additional place for the temporary value. In the same way as above we rename the `Elt` to `PcLabel` and the outcome back to `PcState`.

```
mod! PCSTATE {
 pr(3TUPLE(NAT, NAT, PCLABEL{sort Elt -> PcLabel})*{sort 3Tuple -> PcState})
}
```

As we will have an arbitrary set of PCs, we define the multi-set of all PC states, by instatiating the multi-set from above with the just defined `PcState` sort, and rename the result to `PcStates`.

```
mod! PCSTATES {
  pr(MULTISET(PCSTATE{sort Elt -> PcState})*{sort MultiSet -> PcStates})
}
```

Finally, the state of the whole system is declared as a pair of the cloud state and the pc states.

```
mod! STATE {
  pr(PAIR(CLSTATE{sort Elt -> ClState},PCSTATES{sort Elt -> PcStates})*
       {sort Pair -> State})
}
```

The final part is to specify transitions. We have described the protocol by a state machine, and the following transitions will model the transitions in this machine.

The first transition is the initialization of the syncronization by reading the cloud value, saving it into the local register, and both partners go into busy state.

Note that, since we have declared multi-set as commutative and associative, we can assume that the first element of the multi-set is actually the one we are acting on.

Transitions are different from axioms in the sense that the do not state that two terms are the same, but only that one terms can change into another.

```
mod! GETVALUE { pr(STATE)
  trans[getvalue]:
   < < ClVal:Nat , idlecl > , ( << PcVal:Nat ; OldClVal:Nat ; idlepc >> S:PcStates ) >
     =>
     < < ClVal , busy > , ( << PcVal ; ClVal ; gotvalue >> S ) > .
}
```

The next transition is the critical part, the update of the side with the lower value. Here we are using the built-in `if ... then ... else ... fi` operator.

```
mod! UPDATE {
  pr(STATE)
  trans[update]:
   < < ClVal:Nat , busy > , ( << PcVal:Nat ; GotClVal:Nat ; gotvalue >> S:PcStates ) >
     =>
       if PcVal <= GotClVal then
     < < ClVal , busy > , ( << GotClVal ; GotClVal ; updated >> S ) >
       else
     < < PcVal , busy > , ( << PcVal ; PcVal ; updated >> S ) >
       fi .
}
```

The last transition is sending the both sides of the syncronization into the idle states.

```
mod! GOTOIDLE {
  pr(STATE)
  trans[gotoidle]:
   < < ClVal:Nat , busy > , ( << PcVal:Nat ; OldClVal:Nat ; updated >> S:PcStates ) >
     =>
     < < ClVal , idlecl > , ( << PcVal ; OldClVal ; idlepc >> S ) > .
}
```

This completes the complete specification of the protocol, and we are defining a module `CLOUD` that collects all that.

```
mod! CLOUD { pr(GETVALUE + UPDATE + GOTOIDLE) }
```

Verification

Chapter 4

Gory Details

This chapter presents all syntactic elements of CafeOBJ as well as several meta-concepts in alphabetic order. Concepts are cross-linked for easy accessibility.

?

lists all top-level commands. The ? can be used after many of the top-level commands to obtain help.

**, **>

Starts a comment which extends to the end of the line. With the additional > the comment is displayed while evaluated by the interpreter.

Related: -- comments

--, -->

Starts a comment which extends to the end of the line. With the additional > the comment is displayed while evaluated by the interpreter.

Related: ** comments

==

The predicate == is a binary operator defined for each visible sort and is defined in terms of evaluation. That is, for ground terms `t` and `t'` of the same sort, `t == t'` evaluates to `true` iff terms reduce to a common term.

=/=

Negation of the predicate ==.

apply <action> [<subst>] <range> <selection>

Applies one of the following actions `reduce`, `exec`, `print`, or a rewrite rule to the term in focus.

`reduce, exec, print` the operation acts on the (sub)term specified by `<range>` and `<selection>`.

rewrite rule in this case a rewrite rule spec has to be given in the following form:

```
[+|-][<mod_name>].<rule-id>
```

where `<mod_name>` is the name of a module, and `<rule-id>` either a number n - in which case the n. equation in the current module is used, or the label of an equation. If the `<mod_name>` is not given, the equations of the current module are considered. If the leading + or no leading character is given, the equation is applied left-to-right, which with a leading – the equation is applied right-to-left.

The `<subst>` is of the form

```
with { <var_name> = <term> } +,
```

and is used when applying a rewrite rule. In this case the variables in the rule are bound to the given term.

`<range>` is either `within` or `at`. In the former case the action is applied at or inside the (sub)term specified by the following selection. In the later case it means exactely at the (sub)term.

Finally, the `<selection>` is an expression

```
<selector> { of <selector> } *
```

where each `<selector>` is one of

`top, term` Selects the whole term

`subterm` Selects the pre-chosen subterm (see choose)

`( <number_list> )` A list of numbers separated by blanks as in `(2 1)` indicates a subterm by tree search. `(2 1)` means the first argument of the second argument.

`[ <number1> .. <number2> ]` This selector can only be used with associative operators. It indicates a subterm in a flattened structure and selects the subterm between and including the two numbers given. `[n .. n]` can be abbreviated to `[n]`.

Example: If the term is `a * b * c * d * e`, then the expression `[2 .. 4]` selects the subterm `b * c * d`.

{ <number_set> } This selector can only be used with associative and commutative operators. It indicates a subterm in a multiset structure obtained from selecting the subterms at position given by the numbers.

Example: If the operator `_*_` is declared as associative and commutative, and the current term is `b * c * d * c * e`, then then the expression `{2, 4, 5}` selects the subterm `c * c * e`.

Related: `choose` `start`

axioms { <decls> }

Block enclosing declarations of variables, equations, and transitions. Other statements are not allowed within the `axioms` block. Optional structuring of the statements in a module.

Related: `signature` `imports` `var` `eq` `trans`

bceq [<label-exp>] <term> = <term> if <boolterm> .

Alias: `bcq`

Defines a behaviour conditional equation. For details see `ceq`.

Related: `eq` `ceq` `beq`

beq [<label-exp>] <term> = <term> .

Defines a behaviour equation. For details see `eq`.

Related: `eq` `ceq` `bceq`

bctrans [<label-exp>] <term> => <term> if <boolterm> .

Defines a behaviour conditional transition. For details see `ctrans`.

Related `trans` `ctrans` `btrans`

bop <op-spec> : <sorts> -> <sort>

Defines a behavioural operator by its domain, codomain, and the term construct. `<sorts>` is a space separated list of sort names containing *exactely* one hidden sort. `<sort>` is a single sort name.

For `<op-spec>` see the explanations of `op`.

Related: `op`

bpred <op-spec> : <sorts>

Short hand for `op <op-spec> : <sorts> -> Bool` defining a behavioural predicate.

Related: `op bop pred`

breduce [in <mod-exp> :] <term> .

Alias: `bred`

Reduce the given term in the given module, if `<mod-exp>` is given, otherwise in the current module.

For `breduce` equations, possibly conditional, possibly behavioural, are taken into account for reduction.

Related: `execute reduce`

btrans [<label-exp>] <term> => <term> .

Defines a behaviour transition. For details see `trans`.

Related `trans ctrans bctrans`

cd <dirname>

Change the current working directory, like the Unix counterpart. The argument is necessary. No kind of expansion or substitution is done.

Related: `pwd ls`

ceq [<label-exp>] <term> = <term> if <boolterm> .

Defines a conditional equation. Spaces around the `if` are obligatory. `<boolterm>` needs to be a Boolean term. For other requirements see `eq`.

Related: eq beq bceq

check <options>

This command allows for checking of certain properties of modules and operators.

check regularity <mod_exp> Checks whether the module given by the module expression

<mod_exp> is regular.

check compatibility <mod_exp> Checks whether term rewriting system of the module given by the

module expression <mod_exp> is compatible, i.e., every application of every rewrite rule to every well-formed term results in a well-formed term. (This is not necessarily the case in order-sorted rewriting!)

check laziness <op_name> Checks whether the given operator can be evaluated lazily. If not

<op_name> is given, all operators of the current module are checked.

Related: regularize

choose <selection>

Chooses a subterm by the given <selection>. See apply for details on <selection>.

Related: apply start strat in operator attributes

close

This command closes a modification of a module started by open.

Related: open

comments

The interpreter accepts the following strings as start of a comment that extends to the end of the line: --, -->, **, **>.

The difference in the variants with > is that the comment is displayed when run through the interpreter.

Related: ** --

ctrans [<label-exp>] <term> => <term> .

Defines a conditional transition. For details see trans and ceq.

Related trans btrans bctrans

describe <something>

like the show command with more details. See describe ? for the possible set of invocations.

Related: show

eof

Terminates reading of the current file. Allows for keeping untested code or documentations below the eof mark. Has to be on a line by itself without leading spaces.

eq [<label-exp>] <term> = <term> .

Declares an axiom, or equation.

Spaces around the = are necessary to separate the left from the right hand side. The terms given must belong to the same connected component in the graph defined by the sort ordering.

In simple words, the objects determined by the terms must be interpretable as of the same sort.

One can give an equation a name by providing an optional <label-exp> which is:

[<label-name>] :

Warning: The square brackets here are *not* specifying optional components, but syntactical elements. Thus, a named equation can look like:

eq[foobar] : foo = bar .

Related: ceq beq bceq

execute [in <mod-exp> :] <term> .

Alias: exec

Reduce the given term in the given module, if <mod-exp> is given, otherwise in the current module.

For execute equations and transitions, possibly conditional, are taken into account for reduction.

Related: breduce reduce

extending (<modexp>)

Alias: ex

imports the object specified by modexp into the current module, allowing models to be inflated, but not collapsing. See module expression for format of modexp.

Related: including protecting using

imports { <import-decl> }

Block enclosing import of other modules (protecting etc). Other statements are not allowed within the imports block. Optional structuring of the statements in a module.

Related: signature axioms extending including protecting using

including (<modexp>)

Alias: in

imports the object specified by modexp into the current module. TODO what are the consequences for the models? TODO See module expression for format of modexp.

Related: extending protecting using module expression

input <pathname>

requests the system to read the file specified by the pathname. The file itself may contain input commands. CafeOBJ reads the file up to the end, or until it encounters a line that only contains (the literal) eof.

instantiation of parametrised modules

Parametrized modules allow for instantiation. The process of instantiation binds actual parameters to formal parameters. The result of an instantiation is a new module, obtained by replacing occurrences of parameter sorts and operators by their actual counterparts. If, as a result of instantiation, a module is imported twice, it is assumed to be imported once and shared throughout.

Instantiation is done by

<module_name> (<bindings>)

where <module_name> is the name of a parametrized module, and <bindings> is a comma-separated list of binding constructs.

using declared views you may bind an already declared view to a parameter:

```
<parameter> <= <view_name>
```

If a module `M` has a parameter `X :: T` and a view `V` from `T` to `M'` is declared, `V` may be bound to `X`, with the effect that

1. The sort and operator names of `T` that appear in the body of `M` are replaced by those in `M'`, in accordance with `V`,
2. The common submodules of `M` and `M'` are shared.

using ephemeral views In this case the view is declared and used at the same time.

```
<parameter> <= view to <mod_name> { <view_elements> }
```

See `view` for details concerning `<view_elements>`. The `from` parameter in the `view` declaration is taken from `<parameter>`.

To make notation more succinct, parameters can be identified also by position instead of names as in

```
<mod_name> ( <view_name>, <view_name> )
```

which would bind the `<view_name>`s to the respective parameters of the parametrized module `<mod_name>`.

This can be combined with the ephemeral defintion of a view like in the following example (assume `ILIST` has two parameters):

```
module NAT-ILIST {
  protecting ( ILIST(SIMPLE-NAT { sort Elt -> Nat },
                     DATATYPE   { sort Elt -> Data }) )
}
```

`let <identifier> = <term> .`

Using `let` one can define aliases, or context variables. Bindings are local to the current module. Variable defined with `let` can be used in various commands like `reduce` and `parse`.

Although `let` defined variable behave very similar to syntactic shorthands, they are not. The right hand side `<term>` needs to be a fully parsable expression.

`ls <pathname>`

lists the given `pathname`. Argument is obligatory.

Related: `cd` `pwd`

make <mod_name> (<mod_exp>)

This commands defines a new module <mod_name> by evaluating the module expression <mod_exp>.

Related: module expressions

match <term_spec> to <pattern> .

Matches the term denoted by <term_spec> to the pattern. <term_spec> is either top or term for the term set by the start command; subterm for the term selected by the choose command; it has the same meaning as subterm if choose was used, otherwise the same meaning as top, or a normal term expression.

The given <pattern> is either rules, -rules, +rules, one of these three prefixed by all, or a term. If one of the rules are given, all the rules where the left side (for +rules), the right side (for -rules), or any side (for rules) matches. If the all (with separating space) is given all rules in the current context, including those declared in built-in modules, are inspected.

If a term is given, then the two terms are matched, and if successful, the matching substitution is printed.

module[!|*] <modname> [(<params>)] [<principal_sort_spec>] { mod_elements ... }

Alias: mod

defines a module, the basic building block of CafeOBJ. Possible elements are declarations of

- import - see protecting, extending, including, using
- sorts - see sort declaration
- records - TODO delete?
- variable - see var
- equation - see op, eq, ceq, bop, beq, bceq
- transition - see trans, ctrans, btrans, bctrans

modname is an arbitrary string.

module* introduces a loose semantic based module.

module! introduces a strict semantic based module.

module introduces a module without specified semantic type.

If params are given, it is a parametrized module. See parametrized module for more details.

If `principal_sort_spec` is given, it has to be of the form `principal-sort <sortname>` (or `p-sort <sortname>`). The principal sort of the module is specified, which allows more concise `view`s from single-sort modules as the sort mapping needs not be given.

module expression

In various syntax elements not only module names itself, but whole module expressions can appear. A typical example is

```
open <mod_exp> .
```

which opens a module expression. The following constructs are supported:

module name using the name of a module

renaming `<mod_exp> * { <mappings> }`

This expressions describes a new module where sort and/or operators are renamed. `<mappings>` are like in the case of `view` a comma separated list of mappings of either sorts (`sort` and `hsort`) or operators (`op` and `bop`). Source names may be qualified, while target names are not, they are required to be new names. Renaming is often used in combination with instantiantion.

summation `<mod_exp> + <mod_exp>`

This expression describes a module consisting of all the module elements of the summands. If a submodule is imported more than once, it is assumed to be shared.

parametrized module

A module with a parameter list (see `module`) is a parametrized module. Parameters are given as a comma (`,`) separated list. Each parameter is of the form `[ <import_mode> ] <param_name> :: <module_name>` (spaces around `::` are obligatory).

The parameter's module gives minimal requirements on the module instantiation.

Within the module declaration sorts and operators of the parameter are qualified with `.<parameter_name>` as seen in the example below.

Example:

```
mod* C {
  [A]
  op add : A A -> A .
}
mod! TWICE(X :: C) {
  op twice : A.X -> A.X .
  eq twice(E:A.X) = add.X(E,E) .
}
```

Related: qualified sort etc

on-the-fly declarations

Variables and constants can be declared *on-the-fly* (or *inline*). If an equation contains a qualified variable (see qualified term), i.e., `<name>:<sort-name>`, then from this point on *within* the current equation only `<name>` is declared as a variable of sort `<sort-name>`.

It is allowed to redeclare a previously defined variable name via an on-the-fly declaration, but as mentioned above, not via an explicit redeclaration.

Using a predeclared variable name within an equation first as is, that is as the predeclared variable, and later on in the same equation with an on-the-fly declaration is forbidden. That is, under the assumption that `A` has been declared beforehand, the following equation is *not* valid:

```
eq foo(A, A:S) = A .
```

On-the-fly declaration of constants are done the same way, where the `<name>` is a constant name as in \a:Nat‘. Using this construct is equivalent to defining an operator

```
op <name> : -> <sort>
```

or in the above example, `op a : -> Nat .` These constant definitions are quite common in proof scores.

Related: `var`

op <op-spec> : <sorts> -> <sort> { <attribute-list> }

Defines an operator by its domain, codomain, and the term construct. `<sorts>` is a space separated list of sort names, `<sort>` is a single sort name. `<op-spec>` can be of the following forms:

prefix-spec the `<op-spec>` does not contain a literal _: This defines a normal prefix operator with domain `<sorts>` and codomain `<sort>`

Example: `op f : S T -> U`

mixfix-spec the `<op-spec>` contains exactely as many literal _ as there are sort names in `<sorts>`: This defines an arbitrary mixfix (including postfix) operator where the arguments are inserted into the positions designated by the underbars.

Example: `op _+_ : S S -> S`

For the description of `<attribute-list>` see the entry for operator attributes.

open <mod_exp> .

This command opens the module specified by the module expression `<mod_exp>` and allows for declaration of new sorts, operators, etc.

Related: `close` module expression `select`

operator attributes

In the specification of an operator using the op (and related) keyword, attributes of the operator can be specified. An `<attribute-list>` is a space-separate list of single attribute definitions. Currently the following attributes are supported

`associative` specifies an associative operator, alias `assoc`

`commutative` specifies a commutative operator, alias `comm`

`itempotence` specifies an idempotent operator, alias `idem`

`id: <const>` specifies that an identity of the operator exists and that it is `<const>`

`prec: <int>` specifies the parsing precedence of the operator, an integer . Smaller precedence values designate stronger binding. See operator precedence for details of the predefined operator precedence values.

`l-assoc` and `r-assoc` specifies that the operator is left-associative or

right-associative

`constr` specifies that the operator is a constructor of the coarity sort. (not evaluated at the moment)

`strat: ( <int-list> )` specifies the evaluation strategy. Each integer in the list refers to an argument of the operator, where `0` refers to the whole term, `1` for the first argument, etc. Evaluation proceeds in order of the `<int-list>`. Example:

```
op if_then_else_fi : Bool Int Int -> Int { strat: (1 0) }
```

In this case the first argument (the boolean term) is tried to be evaluated, and depending on that either the second or third. But if the first (boolean) argument cannot be evaluated, no evaluation in the subterms will appear.

Using negative values allows for lazy evaluation of the corresponding arguments.

Remarks:

- Several operators of the same arity/coarity can be defined by using `ops` instead of `op`:

  ```
  ops f g : S -> S
  ```

  For the case of mixfix operators the underbars have to be given and the expression surrounded by parenthesis:

  ```
  ops (_+_) (_*_) : S S -> S
  ```

- Spaces *can* be part of the operator name, thus an operator definition of `op foo op : S -> S` is valid, but not advisable, as parsing needs hints.

- A single underbar cannot be an operator name.

Related: bop

operator precedence ## {opprec}

TODO list the rules for operator precedence

`parse [ in <mod-exp> : ] <term> .`

Tries to parse the given term within the module specified by the module expression `<mod-exp>`, or the current module if not given, and returns the parsed and qualified term.

In case of ambiguous terms, i.e., different possible parse trees, the command will prompt for one of the trees.

Related: qualified term

`pred <op-spec> : <sorts>`

Short hand for `op <op-spec> : <sorts> -> Bool` defining a predicate.

Related: `op` `bpred`

`protect <module-name>`

Protect a module from being overwritten. Some modules vital for the system are initially protected. Can be reversed with `unprotect`.

Related: `unprotect`

`protecting ( <modexp> )`

Alias: `pr`

imports the object specified by `modexp` into the current module, preserving all intended models as they are. See `module expression` for format of `modexp`.

Related: `extending` `using` `including`

`provide <feature>`

discharges a feature requirement: once `provided`, all the subsequent `require`ments of a feature are assumed to have been fulfilled already.

Related: `require`

pwd

Prints the current working directory.

Related: cd ls

qualified sort/operator/parameter

CafeOBJ allows for using the same name for different sorts, operators, and parameters. One example is declaring the same sort in different modules. In case it is necessary to qualify the sort, operator, or parameter, the intended module name can be affixed after a literal .: <name>.<modname>

Example: In case the same sort Nat is declared in both the module SIMPLE-NAT and PANAT, one can use Nat.SIMPLE-NAT to reference the sort from the former module.

Furthermore, a similar case can arise when operators of the same name have been declared with different number of arguments. During operator renaming (see view) the need for qualification of the number of parameters might arise. In this case the number can be specified after an affixed /: <opname>/<argnr>

Related: parametrized module qualified term

qualified term

In case that a term can be parsed into different sort, it is possible to qualify the term to one of the possible sorts by affixing it with : <sort-name> (spaces before and after the : are optional).

Example: 1:NzNat 2:Nat

Related: parse

reduce [in <mod-exp> :] <term> .

Alias: red

Reduce the given term in the given module, if <mod-exp> is given, otherwise in the current module.

For reduce only equations and conditional equations are taken into account for reduction.

Related: execute breduce

regularize <mod-name>

Regularizes the signature of the given module, ensuring that every term has exactely one minimal parse tree. In this process additional sorts are generated to ensure unique least sort of all terms.

Modules can be automatically regularized by the interpreter if the `regularize signature` switch is turn to `on`:

```
set regularize signature on
```

TODO – should we give more details here – unclear to me.

require <feature> [<pathname>]

requires a feature, which usually denotes a set of module definitions. Given this command, the system searches for a file named the feature, and read the file if found. If a pathname is given, the system searches for a file named the pathname instead.

Related: `provide`

restore <pathname>

restores module definitions from the designated file `pathname` which has been saved with the `save` command. `input` can also be used but the effects might be different.

TODO – should we keep the different effects? What is the real difference?

Related: `input` `save` `save-system`

save <pathname>

saves module definitions into the designated file `pathname`. File names should be suffixed with `.bin`.

`save` also saves the contents of prelude files as well as module definitions given in the current session.

Related: `input` `restore` `save-system`

save-system <pathname>

dumps the image of the whole system into a file. This is functionality provided by the underlying Common Lisp system and might carry some restrictons.

Related: `input` `save` `restore`

select <mod_exp> .

Selects a module given by the module expression <mod_exp> as the current module. All further operations are carried out within the given module. In constrast to open this does not allow for modification of the module, e.g., addition of new sorts etc.

Related: open module expression

set <name> [option] <value>

Depending on the type of the switch, options and value specification varies. Possible value types for switches are boolean (on, off), string ("value"), integers (5434443), lists (lisp syntax).

For a list of all available switches, use set ?. To see the current values, use show switches. To single out two general purpose switches, verbose and quiet tell the system to behave in the respective way.

Related: show switches

show <something>

The show command provides various ways to inspect all kind of objects of the CafeOBJ language. For a full list call show ?.

Some of the more important (but far from complete list) ways to call the show command are:

- show [<modexp>] - describes the current modules of the one specified as argument
- show switches - lists all possible switches
- show <term> - displays a term, posible in tree format

See the entry for switches for a full list.

Related: switches describe

signature { <sig-decl> }

Block enclosing declarations of sorts and operators. Other statements are not allowed within the signature block. Optional structuring of the statements in a module.

Related: axioms imports sort op

sort declaration

CafeOBJ supports two kind of sorts, visible and hidden sorts. Visible sorts are introduced between `[` and `]`, while hidden sorts are introduced between `*[` and `]*`.

```
[ Nat ]
*[ Obs ]*
```

Several sorts can be declared at the same time, as in `[ Nat Int ]`.

Since CafeOBJ is based on order sorting, sorts can form a partial order. Definition of the partial order can be interleaved by giving

```
[ <sorts> < <sorts> ]
```

Where `sorts` is a list of sort names. This declaration defines an inclusion relation between each pair or left and right sorts.

Example:

```
[ A B , C D < A < E, B < D ]
```

defines five sorts A,…,E, with the following relations: `C < A`, `D < A`, `A < E`, `B < D`.

switches

The following list is the full set of switches at the time of writing:

TODO subsections of all the switches with explanations for each one

```
trace  whole           off
trace                  off
step                   off
memo                   on
always  memo           off
clean  memo            off
statistics|stats       on
rewrite|rwt  limit     = not specified
stop  pattern          = not specified
reduce  conditions     off
exec  trace            off
exec  limit            = 536870911
exec  normalize        on
include  BOOL          on
include  RWL           on
include  FOPL-CLAUSE   on
auto  context          off
```

```
accept  term            on
regularize|reg  signature  off
check  import           off
check  regularity       off
check  coherency        off
check  sensible         off
check  compatibility    off
check  builtin          on
select  term            off
verbose                 off
quiet                   off
all  axioms             off
show  mode              = :cafeobj
show  var sorts         off
print  mode             = :normal
libpath           = ("/usr/local/cafeobj-1.4/lib" "/usr/local/cafeobj-1.4/exs")
tram|compiler  path     = "tram"
tram|compiler  options  = ""
print  depth            = not specified
accept  =*= proof       off
find  all rules         off
```

Related: set show

start <term> .

Sets the focus onto the given term <term> of the currently opened module or context. Commands like apply, choose, or match will then operate on this term.

Related: apply choose match

tram <options>

TODO - do we have a tram compiler still available???

trans [<label-exp>] <term> => <term> .

Defines a transition, which is like an equation but without symmetry.

See eq for specification of requirements on <label-exp> and the terms.

TODO: should we write more here

unprotect <module-name>

Remove overwrite protection from a module that has been protected with the `protect` call. Some modules vital for the system are initially protected.

Related: `protect`

using (<modexp>)

Alias: `us`

imports the object specified by `modexp` into the current module without any restrictions on the models. See `module expression` for format of `modexp`.

Related: `extending` `including` `protecting`

var <var-name> : <sort-name>

Declares a variable `<var-name>` to be of sort `<sort-name>`. The scope of the variable is the current module. Redeclarations of variable names are not allowed. Several variable of the same sort can be declared at the same time using the `vars` construct:

```
vars <var-name> ... : <sort-name>
```

Related: `op` qualified term on-the-fly

on-the-fly declarations

Variables and constants can be declared *on-the-fly* (or *inline*). If an equation contains a qualified variable (see qualified term), i.e., `<name>:<sort-name>`, then from this point on *within* the current equation only `<name>` is declared as a variable of sort `<sort-name>`.

It is allowed to redeclare a previously defined variable name via an on-the-fly declaration, but as mentioned above, not via an explicit redeclaration.

Using a predeclared variable name within an equation first as is, that is as the predeclared variable, and later on in the same equation with an on-the-fly declaration is forbidden. That is, under the assumption that `A` has been declared beforehand, the following equation is *not* valid:

```
eq foo(A, A:S) = A .
```

On-the-fly declaration of constants are done the same way, where the `<name>` is a constant name as in \a:Nat‘. Using this construct is equivalent to defining an operator

```
op <name> : -> <sort>
```

or in the above example, `op a : -> Nat .` These constant definitions are quite common in proof scores.

Related: `var`

`view <name> from <modname> to <modname> { <viewelems> }`

A view specifies ways to bind actual parameters to formal parameters (see parametrized module). The view has to specify the mapping of the sorts as well as the operators.

The `<viewelems>` is a comma-separated list of expressions specifying these mappings:

```
sort <sortname> -> <sortname>
hsort <sortname> -> <sortname>
op <opname> -> <opname>
bop <opname> -> <opname>
```

and also can contain variable declarations.

Infix operators are represented as terms containing the operator with either literal underscores `_`, or variables: `_*_` or `X * Y`. The `<opname>` can be qualified.

Example: Assume a module `MONOID` with sort `M` and ops `e` and `*` are given, and another `SIMPLE-NAT` with sort `Nat` and operators `0` and `+` (with the same arity). Then the following expression constitutes a view:

```
view NAT-AS-MONOID from MONOID to SIMPLE-NAT {
  sort M -> Nat,
  op   e -> 0,
  op _*_ -> _+_
}
```

In specifying views some rules can be omitted:

1. If the source and target modules have common submodules, all the sorts and modules decalred therein are assumed to be mapped to themselves;
2. If the source and target modules have sorts and/or operators with identical names, they are mapped to their respective counterparts;
3. If the source module has a single sort and the target has a principal sort, the single sort is mapped to the principal sort.

Related: instantiation

MISSING UNCLEAR

chapter 4.4

TO BE REMOVED DISCUSSION

stop command, can be done with set stop pattern ...

Bibliography

[1] N. Dershowitz and J.-P. Jouannaud. “Rewrite Systems”. In: *Handbook of Theoretical Computer Science, Vol.B: Formal Models and Semantics*. The MIT Press/Elsevier Science Publishers, 1990, pp. 245–320.

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

[3] R. Diaconescu and K. Futatsugi. *Logical Semantics of CafeOBJ*. Tech. rep. IS-RR-96-0024S. Japan Advanced Institute for Science and Technology, 1996.

[4] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specifications 1: Equations and Initial Semantics*. Springer-Verlag, 1985.

[5] J. Goguen and R. Burstall. “Institutions: Abstract Model Theory for Specification and Programming”. In: *Journal of the Association for Computing Machinery* 39 (1992), pp. 95–146.

[6] J. Goguen and R. Diaconescu. “An Oxford Survey of Order Sorted Algebra”. In: *Mathematical Structures in Computer Science* 4 (1994), pp. 363–392.

[7] J. Goguen and G. Malcom. *A Hidden Agenda*. Tech. rep. UCSD, 1998.

[8] J.A. Goguen and J. Meseguer. *Order-Sorted Algebra 1: Equational Deduction for Multiple Inheritance, Polymorphism, Overloading and Partial Operations*. Tech. rep. Technical Report SRI-CSL-89-10. SRI Internationa, 1989.

[9] B. Jacobs and J. Rutten. “A Tutorial on (Co)Algebras and (Co)Induction”. In: *EATCS Bulletin* 62 (1997), pp. 222–259.

[10] J.W. Klop. “Term Rewriting Systems: A Tutorial”. In: *EATCS Bulletin* 32 (1987), pp. 143–182.

[11] J. Meseguer. “Conditional Rewriting Logic: Deduction, Models and Concurrency”. In: *Proc. 2nd International CTRS Workshop*. Lecture Notes in Computer Science 516. 1991, pp. 64–91.

[12] J. Meseguer and J.A. Goguen. “Initiality, induction and computability”. In: *Algebraic Methods in Semantics*. Cambridge University Press, 1984, pp. 459–541.