CafeOBJ Reference Manual

Toshimi Sawada, Kokichi Futatsugi, Norbert Preining

2014-06-19

Contents

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 targeting 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 presentation 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. The CafeOBJ distribution also includes a *user manual*. This user manual is slightly outdated with respect to the current status of the language, but is targeting those without and prior knowledge of CafeOBJ.

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.

1.1 Background of CafeOBJ

CafeOBJ is an algebraic specification and verification language. Although it can be employed for all kind of programming (since it is Turing complete), the main target are algebraic specification of software systems. This includes programs, protocols, and all kind of interaction specifications. In addition to being a specification language, it is also a *verification* language, that is, a specification given in CafeOBJ can be verified within the same language environment.

Specification here means that we are trying to describe the inner workings of a software system in a mathematical way, while *verification* means that we give a mathematical proof of certain properties. 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 characterize 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.

CafeOBJ is based on three extensions to the basic many-sorted equational logic:

Order-sorted logic In addition to having different sorts (similar to types in other programming languages), these sorts can be ordered, or in other words, one sort can be a subset of another sort: Take for example the number stack: CafeOBJ allows for the provision of natural numbers, which are part of the rational numbers, which are part of the real numbers. This concept allows for operator inheritance and overloading.

Behavioral logic Algebraic modeling is often based on constructors, i.e., all terms under discussion are built up from given operations, and equality can be decided via an equational theory. While being very successful, it is often necessary to model infinite objects (like data streams), which cannot be achieved in this way. CafeOBJ includes *behavioral logic* and the respective *hidden sorts* as methodology to model infinite objects which identity is defined via behavior instead of the equational theory.

Rewriting logic Aim of a algebraic specification and verification is to give a formal proof of correctness. CafeOBJ contains order-sorted term rewriting as operational semantics, which allows for *execution of proof scores*, CafeOBJ code which forms a proof of the required properties.

There is a wide range of literature on all of these subjects for the interested reader in search for theoretical background. We refer the reader to [1] as a starting point.

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.

2.1 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`.

2.2 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`

2.3 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`.

2.4 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.

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.

Besides giving an example of a specification and verification, we also try to explain several of the most important concepts in CafeOBJ using rather simple examples.

3.1 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

3.2 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 last equation in each models provides a definition of equality by using the *behavioral* equality ==. 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: empt
}
```

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.clstat
}
```

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 -> PcS
}
```

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 -> PcSt
}
```

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) }
```

3.3 Verification

Aim of the verification is to show *correctness* in the sense that no two PCs are at the same time in the busy state. The idea of the proof is to show using induction on the length of transition sequences from initial states to reachable states, that for all reachable states this property is fulfilled.

More specific, we give a characterization of initial states, and show that for initial states the property holds (base case of the induction). Then we show that for all possible transitions, if the target property holds at the beginning of the transition, it also holds at the end of the transition.

Combining this with a (meta-level) induction proof on the length of transition sequences, we show that the target property holds for all reachable states.

Like with loop invariants in other verification schemes, it turns out that a single target property, the exclusion property mentioned above, does not suffice to hold over transitions, i.e., act as transition invariant. Thus, we have to extended it with additional properties.

The first part of this mini-tutorial on the specification of CloudSync contained the full code, but in the following we will, due to space reasons, only include partial code. The latest version of the CloudSync code can be obtained from [2].

But let us start with the definition of predicates for the initial states. The first step is to define some elementary functions on states, counting how many PCs are in a certain state:

```
mod! STATEfuncs {
  pr(NAT + STATE)
  -- no pc in gotvalue state
  pred zero-gotvalue : State .
  pred zero-updated : State .
  ...
}
```

We are collecting a set of predicates, indicated by their predicate name, and define `apply` as an operator that checks each single predicate against a state, and forms the conjunct of the results.

```
mod! APPLYPREDS {
  pr(STATE)
  [PredName < PredNameSeq]
  op (_ _) : PredNameSeq PredNameSeq -> PredNameSeq {assoc} .
  op apply : PredNameSeq State -> Bool .
  eq apply(P:PredName PS:PredNameSeq, S:State) = apply(P,S) and app
}
```

Characterization of the initial state is easy, as it only requires that all PCs as well as the cloud is in idle state.

```
mod! INITPREDS {
  ...
  op cl-is-idle-name : -> PredName .
  op pcs-are-idle-name : -> PredName .
  ...
}
```

In the following we define the predicate specifying initial states:

```
mod! INITIALSTATE {
  pr(INITPREDS)
  op init-name : -> PredNameSeq .
  eq init-name = cl-is-idle-name pcs-are-idle-name .
  pred init : State .
  eq init(S:State) = apply(init-name, S) .
}
```

Let us now turn to the most difficult part, that is finding an invariant. This is not a one-shot technique, but mostly iterative. One starts with a set of predicates, and realizes that the proofs don't work out properly, due to some missing properties. Thus, we add new predicates and iterate until the induction proof finally succeeds.

In the following case we ended up with five different predicates that combined worked as invariant:

`cloud-idle-pcs-idle` If the cloud is in the idle state, then all the pcs are also in the idle state.

`pc-clval` If the cloud is in busy state, then the value of the cloud and the value in the temporary storage area of any PCs in the `gotvalue` or `updated` states agree.

`one-active` At most one PC is out of the idle state.

`gotvalue-cloud-value` If a PC is in the `gotvalue` state, then the value saved in the temporary storage area and the one of the cloud agree.

`goal` If a PC is in the `updated` state, then the value of the PC and the value of the cloud agree.

See the mentioned web-page for the full code of these modules.

In addition to the necessity to introduce additional predicates to obtain an invariant, it also often turns out that some properties, or lemmas, have to be stated or proven so that the verification can work out. In our case some properties on `if_then_else_fi` constructs, as well as consequences of rewriting are included in a module `NECESSARYFACTS`.

The final - and one of the most important parts - is the proof of the two properties:

- base case: if a state satisfies the initial state predicate, it also satisfies the invariant: `red init(S) implies invariant(S) .`
- induction step: if a state satisfies the invariant, and we apply a transition, then the next state also satisfies the invariant: ‘`red inv-condition(S, SS) .`

In both cases we cannot work with a general variable `S`, as in this case no rewriting can take place, and we will not obtain true. What has to be done is to provide a covering set of state expressions, i.e., a set of terms such that every possible instance of a state is also an instance of one of these terms. In our case this is quite easy to provide and consists of six different state terms, combining the three possibilities for a PC with two options of states for the cloud:

```
ops s1 s2 s3 s4 t1 t2 t3 t4 : -> State .
eq s1 =  < < N , idlecl > , ( << M ; K ; idlepc   >> PCS ) >  .
eq s2 =  < < N , idlecl > , ( << M ; K ; gotvalue >> PCS ) >  .
eq s3 =  < < N , idlecl > , ( << M ; K ; updated  >> PCS ) >  .
eq t1 =  < < N , busy   > , ( << M ; K ; idlepc   >> PCS ) >  .
eq t2 =  < < N , busy   > , ( << M ; K ; gotvalue >> PCS ) >  .
eq t3 =  < < N , busy   > , ( << M ; K ; updated  >> PCS ) >  .
```

It is easy to see that any arbitrary state term can be obtained as instance of one of these six state terms.

What we then show is that the above properties do hold for each of these terms, and thus for each of the reachable states. In details, we show that:

```
red init(s1) implies invariant(s1) .
red init(s2) implies invariant(s2) .
red init(s3) implies invariant(s3) .
red init(t1) implies invariant(t1) .
red init(t2) implies invariant(t2) .
red init(t3) implies invariant(t3) .
```

all of these expressions reduce to `true`. And furthermore, all of the following expressions, too:

```
red inv-condition(s1, SS) .
red inv-condition(s2, SS) .
red inv-condition(s3, SS) .
red inv-condition(t1, SS) .
red inv-condition(t2, SS) .
red inv-condition(t3, SS) .
```

Unfortunately, in the case of `t2` this didn't turn out to be directly possible, and a further case distinction was necessary to complete the proof.

This concludes the presentation of the CloudSync protocol. We described the cloud protocol using a *state system* and transitions. This is just one way of implementation. There are other approaches to specification using purely term-based expressions that do not use transitions, but equational theory only. One of the strength of CafeOBJ is that it does not require any specific approach to modeling, but allows for freedom in choosing methodology.

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.

4.1 Ctrl-D

4.2 `! <command>`

On Unix only, forks a shell and executes the given `<command>`.

4.3 `#define`

4.4 `**`, `**>`

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, `--`

4.5 `--`, `-->`

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, `**`

4.6 `.`

Do nothing.

4.7 `:apply (<tactic> ...) [to <goal-name>]` ## {#:apply}

TODO

4.8 `:auto` ## {#:auto}

TODO

4.9 `:backward equation` ## {#:backward}

TODO

4.10 `:cp { "[" <label> "]" | "(" <axiom> . ")" } >< { "[" <label> "]" | "(" <axiom> .")" }` ## {#:cp}

TODO

4.11 `:equation` ## {#:equation}

TODO

4.12 `:goal { <axiom> . ... }` ## {#:goal}

TODO

4.13 `:ind on <variable> ... .` ## {#:ind}

TODO

4.14 `:init { "[" <label> "]" | "(" <axiom> "")} "{" <variable> <- <term>; ... "}"` ## {#:init}

TODO

4.15 `:is` ## {#:is}

Boolean expression: `A :is B` where `A` is a term and `B` is a sort. Returns true if `A` is of sort `B`.

4.16 `:lred <term> .` ## {#:red}

TODO

4.17 `:roll back` ## {#:roll}

TODO

4.18 `:rule` ## {#:rule}

TODO

4.19 `:select <goal-name>` ## {#:select}

TODO

4.20 `:verbose { on | off }` ## {#:verbose}

TODO

4.21 `=`

The syntax element = introduces an axiom of the equational theory, and is different from == which specifies an equality based on rewriting.

Related: `eq`, `==`

4.22 `=(n)=>`, `=(n,m)=>`, `=()=>`

See `search predicates`

4.23 `=*=`

The predicate for behavioural equivalence, written `=*=`, is a binary operator defined on each hidden sort.

TODO: old manual very unclear … both about =*= and `accept =*= proof` ??? (page 46 of old manual)

4.24 `=/=`

Negation of the predicate ==.

4.25 `==`

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. This is different from the equational = which specifies the equality of the theory.

4.26 `==>`

This binary predicate is defined on each visible sort, and defines the transition relation, which is reflexive, transitive, and closed under operator application. It expresses the fact that two states (terms) are connected via transitions.

Related: search predicates, `trans`

4.27 `? [<term>]`

Without any argument, lists all top-level commands. With argument gives the reference manual description of `term`. In addition to this, many commands allow for passing `?` as argument to obtain further help.

In case examples are provided for the `<term>`, they can be displayed using `?ex <term>`. In this case the normal help output will also contain an informational message that examples are available.

When called as ?? both documentation and examples are shown.

4.28 `?apropos <term> [<term> ...]`

Searches all available online docs for the terms passed. Terms are separated by white space. Each term is tested independently and all terms have to match. Testing is done either by simple sub-string search, or, if the term looks like a regular expression (Perl style), by regex matching. In case a regex-like term cannot be parsed as regular expression, it is used in normal sub-string search mode.

Note: Fancy quoting with single and double quotes might lead to unexpected problems.

Example

```
CafeOBJ> ?ap prec oper
```

will search for all entries that contain both `prec` and `oper` as substrings. Matching is done as simple sub-string match.

```
CafeOBJ> ?ap foo att[er]
```

will search for entries that contain the string `foo` as well as either the string `atte` or `attr`.

4.29 `[`

Starts a sort declaration. See sort declaration for details.

4.30 `accept =*= proof` switch

TODO missing documentation difficult - see TODO for `=*=`

4.31 `all axioms` switch

Controls whether axioms from included modules are shown during a `show` invocation.

Related: `show`

4.32 `always memo` switch

Turns on memorization of computation also for operators without the `memo` operator attribute.

Related: operator attributes, `memo`

4.33 `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: `start`, `choose`

4.34 `auto context` switch

Possible values: `on` or `off`, default is `off`.

If this switch is `on`, the context will automatically switch to the most recent module, i.e., defining a module or inspecting a module's content will switch the current module.

4.35 `autoload`

4.36 `ax`

(pignose)

4.37 `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: `trans`, `eq`, `var`, `imports`, `signature`

4.38 `bax`

(pignose)

4.39 `bceq [ <op-exp> ] <term> = <term> if <boolterm> .`

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

Related: `beq`, `ceq`, `eq`

4.40 `bctrans [ <label-exp> ] <term> => <term> if <bool> .`

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

Related: `btrans`, `ctrans`, `trans`

4.41 `beq [ <op-exp> ] <term> = <term> .`

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

Related: `bceq`, `ceq`, `eq`

4.42 `bgoal`

(pignose)

4.43 `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`

4.44 `bpred <op-spec> : <sorts>`

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

Related: `pred`, `bop`, `op`

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

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: `reduce`, `execute`

4.46 `brl`

4.47 `brule`

4.48 `bsort`

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

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

Related: `bctrans`, `ctrans`, `trans`

4.50 `btrns`

4.51 `cbred`

4.52 `cd <dirname>`

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

Related: `ls`, `pwd`

4.53 `ceq [ <op-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: `bceq`, `beq`, `eq`

4.54 `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`

4.55 `check <something>` switch

These switches turn on automatic checking of certain properties:

`check coherency` TODO

`check compatibility` see the `check` command

`check import` TODO

`check regularity` see the `check` command

`check sensible` TODO

4.56 `choose <selection>`

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

Related: `strat` in operator attributes, `start`, `apply`

4.57 `clause`

(pignose)

4.58 `clean memo`

Resets (clears) the memo storages of the system. Memorized computations are forgotten.

Related: clean memo switch

4.59 `clean memo` switch

Possible values: `on`, `off`, default `off`.

tells the system to be forgetful.

4.60 `close`

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

Related: `open`

4.61 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: `--`, `**`

4.62 `cond limit` switch

4.63 `cont`

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

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

Related: `bctrans`, `btrans`, `trans`

4.65 `db`

(pignose)

4.66 `dbpred`

(pignose)

4.67 `demod`

(pignose)

4.68 `describe <something>`

Similar to the `show` command but with more details. See `describe ?` for the possible set of invocations.

Related: `show`

4.69 `dirs`

4.70 `dpred`

(pignose)

4.71 `dribble`

4.72 `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.

4.73 `eq [ <op-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.

The optional part `<op-exp>` serves two purposes, one is to give an axiom an identifier, and one is to modify its behaviour. The `<op-exp>` is of the form:

`[ <modifier> <label> ] :`

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

`eq[foobar] : foo = bar .`

The `<modifier>` part is used to change the rewriting behaviour of the axiom. There are at the moment two possible modifiers, namely `:m-and` and `:m-or`. Both make sense only for operators where the arguments come from an associative sort. In this case both modifiers create all possible permutations of the arguments and rewrite the original term to the conjunction in case of `:m-and` or to the disjunction in case of `:m-or` of all the generated terms.

Assume that `NatSet` is a sort with associative constructor modelling a set of natural number, and let

```
  pred p1: Nat .
```

```
ops q1 q2 : NatSet -> Bool .
eq [:m-and]: q1(N1:Nat NS:NatSet) = p1(N1) .
eq [:m-or]:  q2(N1:Nat NS:NatSet) = p1(N1) .
```

In this case an expression like `q1(1 2 3)` would reduce to `p1(1) and p1(2) and p1(3)` (modulo AC), and `q2(1 2 3)` into the same term with `or` instead.

Related: `bceq`, `beq`, `ceq`

4.74 `exec limit` switch

Possible values: integers, default limit 4611686018427387903.

Controls the number of maximal transition steps.

Related: `reduce`

4.75 `exec trace` switch

Possible values: `on off, default`off‘.

controls whether further output is provided during reductions.

Related: `reduce`

4.76 `exec!`

exec! [in :] .

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

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: `reduce`, `breduce`

4.78 `extending ( <modexp> )`

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: `using`, `protecting`, `including`

4.79 `find`

4.80 `find all rules` switch

4.81 `flag`

(pignose)

4.82 `full reset`

Reinitializes the internal state of the system. All supplied modules definitions are lost.

Related: `reset`

4.83 `gendoc <pathname>`

generates reference manual from system's on line help documents, and save it to `pathname`.

4.84 `goal`

(pignose)

4.85 `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: `using`, `protecting`, `including`, `extending`, `axioms`, `signature`

4.86 `include BOOL` switch

Possible values: `on off`, default `on`.

By default a couple of built-in modules are implicitly imported with protecting mode. In particular, BOOL is of practical importance. It defines Boolean operators. It is imported to admit conditional axioms.

This switch allows to disable automatic inclusion of BOOL.

4.87 `include RWL` switch

Possible values: `on off`, default `off`.

This switch allows to disable automatic inclusion of RWL.

4.88 `including ( <modexp> )`

Imports the object specified by `modexp` into the current module.

See `module expression` for format of `modexp`.

Related: module expression, `using`, `protecting`, `extending`

4.89 `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`.

4.90 `inspect`

4.91 instantiation of parameterized modules

Parameterized 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 parameterized 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 parameterized 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 }) )
}
```

4.92 `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.

4.93 `lex`

(pignose)

4.94 `libpath` switch

Possible values: list of strings.

The switch `libpath` contains a list of directories where CafeOBJ searches for include files. Addition and removal of directories can be done with

```
set libpath + <path1>:<path2>:...
set libpath - <path1>:<path2>:...
```

or the full libpath reset by `set libpath <path1>:<path2>:...`

The current directory has a privileged status: It is always searched first and cannot be suppressed.

4.95 `lisp`

Evaluates the following lisp expression.

Example

```
CafeOBJ> lisp (+ 4 5)
(+ 4 5) -> 9
```

4.96 `lispq`

Evaluates the following quoted lisp expression. (TODO ???)

4.97 `list`

(pignose)

4.98 `look up <something>`

TODO (memory-fault on sbcl)

4.99 `ls <pathname>`

lists the given `pathname`. Argument is obligatory.

Related: `pwd`, `cd`

4.100 `make`

4.101 `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.

4.102 memo switch

controls the memorization of computations. The system memorizes evaluations of operators declared with the `memo` operator attribute. Turning this switch off disables all memorization.

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

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

- import - see `protecting`, `extending`, `including`, `using`
- sorts - see `sort declaration`
- 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 parameterized module. See `parameterized 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 `views` from single-sort modules as the sort mapping needs not be given.

4.104 `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.

4.105 `names`

show

4.106 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 similar to defining an operator

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

or in the above example, `op a : -> Nat .`, besides that the on-the-fly declaration of constants, like to one of variables, is only valid in the current context (i.e., term or axiom). These constant definitions are quite common in proof scores.

Related: `var`

4.107 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.

4.108 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: `select`, `module expression`, `close`

4.109 `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.

`memo` tells the system to remember the results of evaluations where the operator appeared. See `memo` switch for details.

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`

4.110 operator precedence

CafeOBJ allows for complete freedom of syntax, in particular infix operators and overloading. To correctly parse terms that are ambigous, all operators have precedence values. These values can be adjusted manually during definition of the operator (see operator attributes). In absence of manual specification of the operator precedence, the values are determined by the following rules:

- standard prefix operators, i.e., those of the form `op f : S1 .. Sk -> S`, receive operator precedence value 0.
- unary operators, i.e., those of the form `op u_ : S1 -> S`, receive precedence 15.
- mix-fix operators with forst and last token being arguments, i.e., those of the form `op _ arg-or-op _ : S1 .. Sk -> S`, receive precedence 41.
- all other operators (constants, operators of the form `a _ b`, etc.) receive precedence 0.

Related: operator attributes

4.111 option

(pignose)

4.112 param

(pignose)

4.113 `parameterized module`

A module with a parameter list (see `module`) is a parameterized 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.

Related: qualified sort

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) .
}
```

4.114 `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`

4.115 `parse normalize` switch

4.116 `popd`

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

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

Related: `bpred`, `op`

4.118 `prelude`

4.119 `print depth` switch

Possible values: natural numbers, default `unlimited`.

Controls to which depth terms are printed.

4.120 `print mode` switch

Possible values: `normal` `fancy` `tree` `s-expr`

Selects one of the print modes.

4.121 `print trs` switch

Possible values: `on` `off`, default `off`

If set to `on`, print the rules used during reduction of `=(_,_)=>+_if_suchThat_{_}`.

Related: `search predicates`

4.122 `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`

4.123 `protecting ( <modexp> )`

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: `including`, `using`, `extending`

4.124 `provide <feature>`

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

Related: `require`

4.125 `pushd`

4.126 `pvar`

(pignose)

4.127 `pwd`

Prints the current working directory.

Related: `ls`, `cd`

4.128 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: `qualified term`, `parameterized module`

4.129 `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).

Related: `parse`

Example

```
1:NzNat 2:Nat
```

4.130 `quiet` switch

Possible values: `on` `off`, default `off`

If set to `on`, the system only issues error messages.

Related: `verbose`

4.131 `quit`

Leaves the CafeOBJ interpreter.

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

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: `breduce`, `execute`

4.133 `reduce conditions` switch

Possible values: `on` `off`, default `off`.

When using `apply` to step through a reduction, this switch allows to turn on automatic reduction of conditions in conditional equations.

Related: `apply`

4.134 `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`.

4.135 `regularize signature` switch

See ‘regularize

4.136 `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`

4.137 `reset`

Restores the definitions of built-in modules and preludes, but does not affect other modules.

Related: `full reset`

4.138 `resolve`

(pignose)

4.139 `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: `save-system`, `save`, `input`

4.140 `rewrite limit` switch

Possible values: positive integers, default not specified.

Allows limiting the number of rewrite steps during a stepwise execution.

Related: `step switch`

4.141 `rl`

4.142 `rule`

4.143 `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: `save-system`, `restore`, `input`

4.144 `save-option`

(pignose)

4.145 `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: `restore`, `save`, `input`

4.146 `scase`

4.147 `search predicates`

CafeOBJ provides a whole set of search predicates, that searches the reachable states starting from a given state, optionally checking additional conditions. All of them based on the following three basic ones:

- `S =(n,m)=>* SS [if Pred]` search states reachable by 0 or more transitions;
- `S =(n,m)=>+ SS [if Pred]` search states reachable by 1 or more transitions;
- `S =(n,m)=>! SS [if Pred]` search states reachable by 0 or more transitions, and require that the reached state is a final state, i.e., no further transitions can be applied.

To allow for conditional transitions, a transition is only considered in the search if `Pred` holds.

The parameters `n` and `m` in these search predicates:

- `n`, a natural number of `*`, gives the maximal number of solutions to be searched. If `*` is given all solutions are searched exhaustively.
- `m`, a natural number but not `*`, gives the maximal depth up to which search is performed.

The predicates return true if there is a (chain of) transitions that fits the parameters (`n`,`m`, and `*`, `+`, `!`) and connects `S` with `SS`.

There are two orthogonal extension to this search perdicate, one adds a `suchThat` clause, one adds a `withStateEq` clause.

S =(n,m)=>* SS [if Pred1] suchThat Pred2 (and similar for ! and +) In this case not only the existence, of a transition sequence is tested, but also whether the predicate `Pred2`, which normally takes `S` and `SS` as arguments, holds.

S =(n,m)=>* SS [if Pred1] withStateEq Pred2 (and similar for ! and +) TODO

These two cases can also be combined into

```
S =(n,m)=>* SS [if Pred1] suchThat Pred2 withStateEq
Pred3
```

4.148 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: `module expression`, `open`

4.149 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: `switches`, `show`

4.150 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: `describe`, `switches`

4.151 `show mode` switch

Possible values for `set show mode <mode>` are `cafeobj` and `meta`.

TODO no further information on what this changes

4.152 `sigmatch`

(pignose)

4.153 `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: `op`, `sort`, `imports`, `axioms`

4.154 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`.

4.155 `sos`

(pignose)

4.156 `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: `match`, `choose`, `apply`

4.157 `statistics` switch

Possible values: `on off`, default `on`.

After each reduction details about the reduction are shown. Information shown are the time for parsing the expression, the number of rewrites and run time during rewriting, and the number of total matches performed during the reduce.

4.158 `step` switch

Possible values: `on off`, default `off`.

With this switch turned on, rewriting proceeds in steps and prompts the user interactively. At each prompt the following commands can be given to the stepper (with our without leading colon `:`):

`help` (`h`, `?`) print out help page
`next` (`n`) go one step
`continue` (`c`) continue rewriting without stepping
`quit` (`q`) leave stepper continuing rewrite
`abort` (`a`) abort rewriting
`rule` (`r`) print out current rewrite rule
`subst` (`s`) print out substitution
`limit` (`l`) print out rewrite limit count

`pattern` `(p)` print out stop pattern
`stop [<term>] .` set (or unset) stop pattern
`rwt [<number>] .` set (or unset) max number of rewrite

Other standard CafeOBJ commands that can be used are `show`, `describe`, `set`, `cd`, `ls`, `pwd`, `lisp`, `lispq`, and (on Unix only) `!`.

4.159 `stop`

4.160 `stop pattern` switch

This command causes reductions to stop when the reductants get to containing subterms that match the given term. If no term is given, this restriction is lifted.

TODO does not work as far as I see – shouldn't the following code fragment stop at the occurrence of `(s 2)`, before rewriting it to the final 3?

```
CafeOBJ> open NAT .

-- opening module NAT.. done.

%NAT> set stop pattern (s 2) .

%NAT> red (s (s (s 0))) .
-- reduce in %NAT : (s (s (s 0))):NzNat
(3):NzNat
(0.000 sec for parse, 3 rewrites(0.000 sec), 3 matches)

%NAT>
```

Related: `step switch`

4.161 switches

Switches control various aspects of the computations and behaviour of CafeOBJ. The current list of switches and their values can be shown with

```
show switches
```

The single switches are described separately in this manual.

Related: `show`, `set`

4.162 `trace [whole]` switch

During evaluation, it is sometimes desirable to see the rewrite sequences, not just the results. Setting the switch `trace whole` will result in the resultant term of each rewrite step being printed. Setting the switch `trace` will result in the display of which rule, substitution, and replacement are used.

4.163 `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.

Transitions and equations server similar, but different purpose. In particular, reductions (both with or without behavior axioms used) do not take transitions into account. Only `exec` also uses transitions. On the other hand, the built-in search predicate searches all possible transitions from a given term.

4.164 `trns`

4.165 `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`

4.166 `using ( <modexp> )`

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

Related: `protecting`, `including`, `extending`

4.167 `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: `on-the-fly`, `qualified term`, `op`

4.168 `verbose` switch

Possible values: `on` `off`, default `off`.

If turn `on`, the system is much more verbose in many commands.

Related: `quiet switch`

4.169 `version`

Prints out the version of CafeOBJ.

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

A view specifies ways to bind actual parameters to formal parameters (see parameterized 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.

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.

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 _*_ -> _+_
}
```

Bibliography

[1] Sh. Iida, J. Meseguer, and K. Ogata, eds. *Specification, Algebra, and Software - Essays Dedicated to Kokichi Futatsugi*. Vol. 8373. LNCS. Springer, 2014. ISBN: 978-3-642-54623-5.

[2] Norbert Preining. *CafeOBJ*. http://www.preining.info/blog/cafeobj/.