CafeOBJ Commands
Quick Reference
(for interpreter version February 2016 v1.5.6)

Notation

Keywords upper in type setter face, when presented in the form like 'x(yz)' it means the keyword 'xyz' can be abbreviated to 'x'. '[something]' means 'something' is optional. | is used for listing alternatives. Slanted face, e.g., \textit{varety} is used when it varies (a metavariable) or is an expression of some language. For example, \texttt{module} is for module expressions and \texttt{term} is for terms (you should know what these are); others should easily be understood by their names and/or from the context.

Starting CafeOBJ interpreter

To enter CafeOBJ, just type its name: cafeobj
‘cafeobj  -help’ will show you a summary of command options.

Leaving CafeOBJ

q(uit) exits CafeOBJ.

Getting Help

Typing ? at the top-level prompt will print out a online help guidance. This is a good starting point for navigating the system. Also try typing \texttt{com}, this shows the list of major toplevel commands.

Escape

There would be a situation that you hit return expecting some feedback from the interpreter, but it does not respond. This occurs when the interpreter expects some more inputs from you thinking preceding input is not yet syntactically complete. If you encounter this situation, first, try type in '.,’ and return. When this does not help, then type in esc(escape key) and return, it will immediately be back to you discarding preceding input and makes a fresh start.

Rescue

Occasionally you may meet a strange prompt \texttt{CHAOS\textasciitilde} (on platform Allegro CommonLisp) or $\texttt{0}$ (on platform SteelBank CommonLisp) after some error messages. This happens when the interpreter caused some internal errors and could not recover from it. There should be some guides printed describing how you can recover from it, please follow them.

Sending interrupt signal (typing \texttt{C-c from keyboard, or if you are in Emacs, some key sequence specific to the mode you are in}) forces the interpreter to break into underlying Lisp, and you will see the same prompt as the above. This might be useful when you feel the interpreter got confused. :q also works for returning to CafeOBJ interpreter from Lisp.

Setting Switches

Switches are for controlling the interpreter’s behaviour in several manner. The general form of setting top-level switch is:

\texttt{set \textit{switch} value}

In the following, the default value of a switch is shown underlined.

<table>
<thead>
<tr>
<th>switch</th>
<th>value</th>
<th>what?</th>
</tr>
</thead>
<tbody>
<tr>
<td>trace whole</td>
<td>on/off</td>
<td>trace top-level rewrite step</td>
</tr>
<tr>
<td>trace</td>
<td>on/off</td>
<td>trace every rewrite step</td>
</tr>
<tr>
<td>step</td>
<td>on/off</td>
<td>stepwise rewriting process</td>
</tr>
<tr>
<td>memo</td>
<td>on/off</td>
<td>enable term memoization</td>
</tr>
<tr>
<td>always memo</td>
<td>on/off</td>
<td>implicitly set ‘memo’ attributes to all user defined operators</td>
</tr>
<tr>
<td>clean memo</td>
<td>on/off</td>
<td>clean up term memo table before normalization</td>
</tr>
<tr>
<td>stats</td>
<td>on/off</td>
<td>show statistics data after reduction</td>
</tr>
<tr>
<td>rwt limit</td>
<td>number</td>
<td>maximum number of rewriting</td>
</tr>
<tr>
<td>stop pattern</td>
<td>[term]</td>
<td>stop rewriting when meets</td>
</tr>
<tr>
<td>reduce conditions</td>
<td>on/off</td>
<td>reduce conditional part in apply command</td>
</tr>
<tr>
<td>verbose</td>
<td>on/off</td>
<td>set verbose mode</td>
</tr>
<tr>
<td>exec trace</td>
<td>on/off</td>
<td>trace concurrent execution</td>
</tr>
<tr>
<td>exec limit</td>
<td>number</td>
<td>limit maximum number of concurrent execution</td>
</tr>
<tr>
<td>***</td>
<td></td>
<td>– switches for system’s behaviour</td>
</tr>
<tr>
<td>include BOOL</td>
<td>on/off</td>
<td>– import BOOL implicitly</td>
</tr>
<tr>
<td>incude RWL</td>
<td>on/off</td>
<td>– import RWL implicitly</td>
</tr>
<tr>
<td>include FOPL-CLAUSE</td>
<td>on/off</td>
<td>– import FOPL-CLAUSE implicitly</td>
</tr>
<tr>
<td>auto context</td>
<td>on/off</td>
<td>change current context in automatic</td>
</tr>
<tr>
<td>reg signature</td>
<td>on/off</td>
<td>regularize module signature in automatic</td>
</tr>
<tr>
<td>check regularity</td>
<td>on/off</td>
<td>perform regularity check of signature in automatic</td>
</tr>
<tr>
<td>check compatibility</td>
<td>on/off</td>
<td>perform compatibility check of TRS in automatic</td>
</tr>
<tr>
<td>quiet</td>
<td>on/off</td>
<td>system mostly says nothing</td>
</tr>
<tr>
<td>all axioms</td>
<td>on/off</td>
<td>– show/display options</td>
</tr>
<tr>
<td>show mode</td>
<td>:cafeobj</td>
<td>print all axioms in ‘sh(e) modexp’ command</td>
</tr>
<tr>
<td>show var sorts</td>
<td>on/off</td>
<td>set syntax of printed modules or views</td>
</tr>
<tr>
<td>print mode</td>
<td>:normal</td>
<td>print variables with sorts</td>
</tr>
<tr>
<td></td>
<td>:fancy</td>
<td>set term priting form</td>
</tr>
<tr>
<td></td>
<td>:tree</td>
<td></td>
</tr>
<tr>
<td></td>
<td>:fancy</td>
<td></td>
</tr>
<tr>
<td></td>
<td>:fancy</td>
<td></td>
</tr>
<tr>
<td>libpath</td>
<td>pathname</td>
<td>set file search path</td>
</tr>
<tr>
<td>print depth</td>
<td>number</td>
<td>maximum depth of terms to be printed</td>
</tr>
</tbody>
</table>

The default value of \texttt{pathname} of \texttt{set libpath} command is ‘\texttt{Cafeobj/home/share/cafeobj/version/lib/}’, where \texttt{version} is a version number of the release, as of this writing, it is 1.5.5, and ‘\texttt{Cafeobj/home/}’ varies depending on the installation options of your interpreter. By default it is ‘\texttt{/usr/local/}’, so it will be ‘\texttt{/usr/local/share/cafeobj1.5.1ib/}’.

The default value of \texttt{number} in ‘\texttt{set rwt limit}’ command is 0 meaning no limit counter of rewriting is specified.

Omitting \texttt{term} in \texttt{set stop pattern} sets the stop pattern to empty, i.e., no term will match to the pattern.
Examing Values of Switches

show switch | print list of available switches
show switch switch | print out the value of the specified switch

Setting Context

select modexp
This sets the context of the interpreter (current module) to the module specified by modexp. It must be written in single line. When you type in modexp, the ';' treated as a line continuation (that is, it is effectively ignored), so that you can type in multiple lines for long module expressions. Note that one or more blank characters are required before ';

Inspecting Module

sh(ow) and desc(ribe) commands print information on a module. In the sequel, we use a meta-variable show which stands for either sh(ow) or desc(ribe). Most of the cases, giving desc(ribe) for show gives you more detailed information.

show modexp | prints a module modexp, giving 'i', as modexp shows the current module
show sorts [modexp] | prints sorts of modexp
show ops [modexp] | prints operators of modexp
show vars [modexp] | prints variables of modexp
show params [modexp] | prints parameters of modexp
show subs [modexp] | prints direct submodules of modexp
show sign [modexp] | prints sorts and ops combined

modexp must be given in one line. The same convention for long module expressions is used as that of select command (see Setting Context above.) If the optional [modexp] is omitted, it defaults to the current module. Optionally supplying all before sorts, ops, axioms, and sign, i.e., desc all ops for an instance) makes printed out information also include import sorts, operators, etc. otherwise it only prints own constructs of the modexp.

The following show commands assume the current module is set to some module.

show sort sort | prints information on sort sort
show op operator | prints information on operator operator
show [all] axioms | prints (all) axioms in the current module
show [all] rules | prints (all) rewrite rules in the current module

For inspecting submodules or parameters, the following show commands are useful:

show param argname | prints information on the parameter argname
show sub n | prints information on the nth direct submodule

argname can be given by position, not by name.

You can see the hierarchy of a module or a sort by the following sh(ow) commands:

sh(ow) module tree modexp | prints pictorial hierarchy of module. specifying , as modexp shows the hierarchy of the current module
sh(ow) sort tree sort | prints hierarchy of sort pictorially

Evaluating Terms

reduce | in modexp | term .
execute | in modexp | term .
reduce reduces a given term term in the term rewriting system derived from modexp. execute is similar to reduce, but it also considers axioms given by transition declarations. In both cases, omitted 'in modexp : ' defaults to the current module.

The result term of reduce and execute is bound to special variables $term and $subterm (see the next section).

Let Variables and Special Variables

let let-variable = term .
let-variable is an identifier. Assuming the current module is set, let binds let-variable to the given term term. Once set, let-variable can be used wherever term can appear.

You can see the list of let bindings by:

sh(ow) let

There are two built-in special variables in the system:

$term bound to the result term of reduce, execute, parse, or start commands.
$subterm bound to the result of choose command

Let variables and special variables belongs to a context, i.e., each context has its own let variables and special variables.

Inspecting Terms

parse | in modexp | term .
parse parses given term term in the module modexp (if omitted, parses in the current module) and prints the result. The result is bound to special variables $term and $subterm.

The following sh(ow) command assumes the current module, and prints the term.

sh(ow) term [let-variable] [tree]
let-variable can be a name of let-variable, $term or $subterm, if omitted the term bound to $term is printed. If optional tree is supplied, it prints the term tree structure. By setting a switch tree horizontal to true, the term tree will be shown horizontally.

Opening/Closing Module

open modexp | opens module modexp
close close the currently opening module

Opening module can be modified, i.e., you can declare new sorts, operators, axioms. You can open only one module at a time.

Applying Rewrite Rules

Start The initial target (entire term) is set by start command.
start term.

This binds two special variables $term and $subterm to term.
Apply `apply` command applies actions to (sub)term of `$$term`.

```plaintext
apply action range selection
```

You specify an action by `action`, and it will be applied to the target (sub)term specified by `selection`.

```plaintext
range is either `within` or `at`: `within` means at or inside the (sub)term specified by the selection, and `at` means exactly at the selection.
```

**Action** `action` can be the followings:

```plaintext
red(uction) reduce the selected term
exec execute the selected term
print print the selected term
rule-spec apply specified rule to the selected term
```

**Rule-Spec** `rule-spec` specifies the rule with possibly substitutions being applied, and given by

```plaintext
[* | -][modexp].[rule-name][substitutions]
```

The first optional `*` specifies the direction of the rule; left to right (if omitted) or right to left (if `-`).

A rule itself is specified by `[modexp].[rule-name]`. This means the rule with name `rule-name` of the module `modexp` (if omitted, the current module). `rule-name` is either a label of a rule or a number which shown by `sh(ow) rules` command (see **Showing Available Rules** below.)

```plaintext
substitution` binds variables that apper in the selected rule before applying it. This has the form

with `variable = term`, ...
```

**Showing Available Rules** To see the list of the rewrite rules, use `sh(ow) [all] rules`

The list of (all, i.e., includes imported rules if the optional `all` is supplied) available rules are printed with each of which being numbered. The number can be used for `rule-name` (see above).

**Selection** `selection` is a sequence of `selector` separated by keyword of specifying (sub)term of `$$term`:

```plaintext
selector { of selector } ...
```

```plaintext
selector
description
top the entire term (`$$term`
 ditto
subterm selects `$$subterm`
(number ...) selects by position
[number .. number] by range in flattened term structure
{ number , ... } subset in flattened term structure
```

**Step by Step Subterm Selection** choose command selects a sub-term of `$$subterm` and reset the `$$subterm` to the selected one.

```plaintext
choose selector
```

**Matching Terms**

```plaintext
match `term-spec to pattern`
```

term-spec specifies the term to be matched with pattern:

```plaintext
term-spec description
```

**Stepper**

If the switch `step` is set to on, invoking `reduce` or `execute` command runs into the term rewriting stepper. The stepper has its own command interpreter loop, where the following stepper commands are available:

```plaintext
? print out available commands.
n(ext) go one step
l(eave) leave stepper continuing rewrite
r(epeat) redisplays previous stepper commands
s(ubmit) prints substitution
l(imit) set rewrite limit counter
p(rint) prints pattern
s(ubstitution) prints substitution
r(write) prints substitution
r(ule) prints current rewrite rule
```

```plaintext
reset system to initial status
full reset recover definitions of built-in modules
```

**Reading In Files**

```plaintext
input file read in CafeOBJ program from file
provide feature provide the feature
require feature [pathname] require feature
```

**Protecting Your Modules**

```plaintext
protect modexp prevent the module from redefinition
unprotect modexp allow module to be redefined
```
Little Semantic Tools

check regularity [modexp] reports the result of regularity check of module
check comatibility [modexp] reports the result of compatibility check of the module

For both commands, omitted modexp will perform the check in the current module.

The following check command assumes the current module:
check laziness [operator]
This checks strictness of operator. If operator is omitted all of the operators declared in the current modules are checked.

Miscellany

ls pathname list contents of directories
cd pathname change working directory of the interpreter
pwd prints working directory
! command fork shell command
ev lisp evaluate lisp expression lisp printing the result
evq lisp evaluate lisp expression lisp