CafeOBJ User's Manual
— ver.1.4 —

Ataru T. Nakagawa Toshimi Sawada Kokichi Futatsugi

Changes from Version 1.3

The main changes from version 1.3 are as follows.

- In addition to Gnu Common Lisp, CMU Common Lisp and Allegro Common Lisp can be used as a platform.
- Faster rewrite engines are now available, and can be invoked from CafeOBJ.
- The predicate ==> (for transition relations) gets a more powerful support. A couple of new switches are added for this reason.
- Behavioural axioms can be used in equational reduction, and an operator attribute `coherent` is added for this purpose.
- A behavioural reduction command is introduced.
- Sort predicates are introduced.

Contents

Introduction

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], [13] (for basics of algebraic specifications), [8], [6] (for order-sorted logic), [7] (for hidden sorts), [10] (for coinduction), [12] (for rewriting logic), [5] (for institutions), and [11], [1] (for term rewriting systems), as primers. The logical aspects of CafeOBJ are explained in detail in [2] and [3]. 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[1].

> **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**[12]. 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.

[1]Some of the authors feel that some of these technical terms are queer. But to avoid confusion, we basically keep to the established terminology.

Legends

Restrictions, particulars, and prejudices of the current implementation shall be highlighted in boxes, as

Warning Woe betide those who do not heed this warning!

while syntactical definitions are shown as

Syntax 1: *title*
```
I am a language construct.
```

The definitions inside the box are by way of notations as follows.

(1) {, } are meta-syntactical brackets.

(2) | separates alternatives.

(3) $*$ means zero or more repetitions of the preceding construct.

(4) $+$ means one or more repetitions.

(5) $*_,$ means zero or more repetitions separated by commas.

(6) $+_,$ means one or more repetitions separated by commas.

(7) [,] enclose optional constructs.

(8) The above symbols, to appear as they are, are enclosed by quotes, as "`{`".

(9) Symbols in the typewriter font appear as they are: `x` is literally `x`. In case this convention is ambiguous, e.g. when a parenthesis appears in isolation, symbols are quoted as in (8).

For example,

`A` $+_,$

represents sequences of one or more `A`'s, separated by commas.

Chapter 1

Dialogue with the System

Availability

You may obtain a complete distribution package from the following sites. In Japan, the relevant files are

```
ftp://ftp.sra.co.jp/pub/lang/CafeOBJ/cafeobj-???.tar.gz
ftp://ftp.sra.co.jp/pub/lang/CafeOBJ/tram-???.tar.gz
ftp://ftp.sra.co.jp/pub/lang/CafeOBJ/brute-???.tar.gz
```

where ??? is a version numbering. Or you can get a system linked from the site

```
http://www.ldl.jaist.ac.jp/cafeobj/
```

Mirror sites in Europe and USA are under consideration.

The current version works under Unix operating systems (usual vernaculars, including various versions of Sun OS, FreeBSD, and Linux, are all okay) or Macintosh, and is implemented on various Common Lisp processors (including Gnu, CMU, Allegro, and MacLisp). But you cannot run, for example, MacLisp on Solaris, for an obvious reason. On the question of which Lisp systems to use on which operating systems, and which Lisp systems to be obtained from where, consult the installation guide (README) in the distribution package.

In principle the system should also work on Windows with Allegro, but there is no guarantee of its behaviours. The implementors have no intention of testing the system under Windows environments.

1.1 Hello, Goodbye

1.1.1 Starting Up

After installation, type "`cafeobj`" to your shell to invoke the system. Then the CafeOBJ interpreter starts up, producing a welcoming message.

```
% cafeobj
-- loading standard prelude
; Loading /mnt1/lang/cafeobj-1.4/prelude/std.bin.

                    -- CafeOBJ system Version 1.4.100203p116 --
                    built: 1984 Dec 24 Mon 3:44:17 GMT
```

```
                    prelude file: std.bin
                             ***
                 1999 Dec 31 Fri 23:59:45 GMT
                        Type ? for help


CafeOBJ>
```

For a moment, ignore the message. "`CafeOBJ>`" is a prompt for your requests. You may type modules, views, parsing instructions, reduction commands, and so on, each of which shall be gradually introduced in the rest of the manual.

As a first experiment, try feeding a trivial module definition (you need not worry about the meaning for now).

```
CafeOBJ> module TEST { [ Elt ] }

-- defining module TEST_* done.
CafeOBJ>
```

Your input is up to the first newline character. The rest is from the system. It shows that the system has accepted the definition, put this module somewhere in its memory, to be available hereafter. You may continue to feed other modules, or check if indeed the module is correctly stored. As a further trial, type

```
CafeOBJ> select TEST

TEST>
```

The command `select` takes as argument a module name. "`TEST>`" is a new prompt. As you see, `TEST` is the name of the module just `select`ed. The system maintains a module, called the *current module*, that is the focus of manipulation at the time. The initial prompt "`CafeOBJ>`" indicates the current module is not yet supplied. Or a module `CafeOBJ` is now current (it is possible, if confusing, to define such a module). Various commands, such as parsing and reduction, are executed under the current module. For more details on current modules, see Section 8.1.1.

1.1.2 Quitting the System

To quit CafeOBJ, type `quit` (or `q` for short) or the end of input character (usually control-D) at a prompt. Then the interpreter terminates and the control is given back to the original shell.

```
CafeOBJ> quit
[Leaving CafeOBJ]
%
```

1.1.3 Emergency — Resumption and Bug Reports

In detecting errors, CafeOBJ growls.

```
CafeOBJ> module ERROR }
[Error]: was expecting the symbol '{' not '}' .

CafeOBJ>
```

The above message starting with "[Error]" indicates that the system detected a syntax error in processing your input.

Due to irrecoverable input errors, or just to mulfunctions, CafeOBJ may stop abnormally, and throw the control away to an error environment.

```
CafeOBJ> ^C
Correctable error: Console interrupt.
Signalled by CAFEOBJ-TOP-LEVEL.
If continued: Type :r to resume execution, or :q to quit to top level.
Broken at SYSTEM:TERMINAL-INTERRUPT.  Type :H for Help.
CHAOS>>
```

The above session was obtained by typing the interrupt code (usually Control-C) at the initial prompt, forcing CafeOBJ to stop willy-nilly. The prompt "CHAOS>>" indicates you plunged into an error environment. The above message may look familiar to you, if you are a user of GCL. The message was from GCL, and the error environment is a GCL error loop[1].

Following the instructions from GCL, you may resume the CafeOBJ session with :q, as

```
CHAOS>>:q

CafeOBJ>
```

There may be irrecoverable cases, when :q (or similar commands of other Common Lisp systems) fails to work. Then try "(cafeobj t)". This is a more robust means to return to CafeOBJ. If, for goodness, you are unable to return to CafeOBJ even then, exit GCL (control-D normally works) and start from scratch.

If the system still works fine, but if you happen to have destroyed some vital module definitions, try

```
CafeOBJ> reset
```

which restores the definitions of built-in modules (Chapter 11) and preludes (Section 1.2.3). This command does not affect other modules. For complete cleanup, use

```
CafeOBJ> full-reset
```

which reinitialises the internal state of the system. But before calling this command, remember that you will lose all the module definitions you have supplied so far. You are warned.

Bug Reports The system is under continual improvement. If you detect any bugs, malfunctions, or just inconveniences, do let us know. The support team is reachable via

```
cafe-bugs@ldl.jaist.ac.jp
```

1.2 Files and Libraries

1.2.1 Reading Files

Not every module is as trivial as TEST, and you will sooooon need library systems for store/recovering module definitions, test cases, and so on. CafeOBJ requires every imported module (cf. Section 2.5) to be defined beforehand, so you need a recursive file reference mechanism. As a minimum, you need the following command.

[1]Error loops of CMU, Allegro, and MacLisp work more or less similarly.

Syntax 1: `input` *command*
`input` *pathname*

`input` (or `in` for short) requests the system to read the file specified by the pathname. The file itself may contain `input` command, and so on, ad nauseum. CafeOBJ reads the file up to eof, or until it encounters a line that only contains (the literal) `eof`.

Syntax 2: `eof` *command*
`eof`

This little command is especially useful when the program is under development. You can supply the system with only those definitions that are up to scratch.

Pathnames follow the conventions of the Unix file system. Both relative and absolute paths are possible, and "~" denotes your home directory. If there is no file with the pathname, the system expands the name with suffixes "`.mod`" or "`.bin`" and re-search the directory. By convention, a file suffixed with "`.mod`" is assumed to contain definitions and commands written in CafeOBJ, while "`.bin`" indicates a file containing internal codes. Normally, you should prepare "`.mod`" files yourself, and leave the system to create "`.bin`" files (see Section 1.2.2).

Some directory traversal commands of Unix are also available.

Syntax 3: `cd` *command*
`cd` *pathname*

Syntax 4: `pwd` *command*
`pwd`

Syntax 5: `ls`*command*
`ls` *pathname*

where pathnames must be those of directories.

Warning The above commands act like their namesakes in Unix shells, but are not exactly the same. The arguments to `cd`, `ls` are not optional, but necessary. Also, the system does not recognise `~bimbo` as the home directory of `bimbo`.

1.2.2 Saving and Restoring

Module definitions may be saved in an internal format, for efficient later retrieval. The commands

Syntax 6: `save` *command*
`save` *pathname*

Syntax 7: `restore` *command*
`restore` *pathname*

save/restore module definitions into/from the designated file. Pathnames must be those of files. To adhere to the system's convention, file names should be suffixed with "`.bin`". In place of `restore` command, you may also use `input` command (Section 1.2.1).

Warning `save` saves the contents of prelude files (see Section 1.2.3) as well as module definitions given during the session. `restore` reestablishes those definitions as they are saved, and the effects may be different from reading the file with `in` command. It is safer to use `restore` command to read `saved` files.

You can also save the entire image of the system, including the state of module definitions, into a dump file, by the following command.

Syntax 8: `save-system` *command*
`save-system` *pathname*

Warning `save-system` dumps the image of Common Lisp, so do not abuse this command. And due to the restictions of Common Lisp systems, this command may not work.

1.2.3 Initialisations and Options

Recall (Section 1.1) that the initial system message contained

```
-- loading standard prelude
; Loading /mnt1/lang/cafeobj-1.4/prelude/std.bin.
```

In starting up, CafeOBJ autoloads a set of modules, called standard preludes. They define frequently used data types and house-keeping data (cf. Chapter 11).

You may initialise the system to your taste, with options and initialisation files. If a file named "`.cafeobj`" is under your home directory, the system reads the file as if you type

```
CafeOBJ> in ~/.cafeobj
```

after the startup. This initialisation is useful to change default values of modes and switches (cf. `set` command, Section 1.3.2).

Various options control the initial state of the system.

-q. This option prohibits reading `.cafeobj`.

-p *pathname*. Instead of the standard prelude file, which is set up during installation, the system reads the given pathname (which must be a file). The suffix convention is applicable here.

+p *pathname*. The sytstem reads the given pathname *in addition to* the standard prelude file.

`-l` *pathname-list*. In processing `input` command, the system searches the pathname list to locate the specified file. (For default search paths, see Section 1.2.5.) All the pathnames must be directories, and the list must be separated by ":".

`+l` *pathname-list*. Pathnames are *added* to the search directory list.

For example,

```
% cafeobj +p /usr/cern/particle/muon.bin
          -l /usr/cern/lib/std:/usr/cern/lib/muon:/usr/home/eva/cafe
```

makes the system (1) read the standard prelude file, (2) read `muon.bin` in addition, (3) set the search directories as given, and (4) read the initialisation file `.cafeobj`, if it exists.

You may also make the system read files by giving them as arguments. For example,

```
% cafeobj lemma1 lemma2
```

has the same effects as typing

```
CafeOBJ> input lemma1
... messages from the system ...
CafeOBJ> input lemma2
... messages from the system ...
```

at the beginning of the session.

Note that, even though "-p" options, file name arguments and ".cafeobj" all cause the system to process definitions and commands, their effects are slightly different. Modules defined in prelude files, whether standard or user-supplied, are treated as preludes. This difference reveals itself when, e.g., `reset` command (Section 1.1) is invoked.

Finally, if you are disinclined to remember all of the above, just memorise the help option.

```
% cafeobj -help
```

lists the available options.

1.2.4 Batch CafeOBJ

If you do not want to pay much attention to the system, you may run the system non-interactively. An option `-batch` causes the system to run on the batch mode, and the system terminates immediately. This is useful in processing a large file. For example,

```
% cafeobj -batch goliath > cafe.out
```

forces the system to read and process `goliath` (or `goliath.bin`, or `goliath.mod`) while you have an afternoon tea.

1.2.5 Module Libraries

As suggested in the previous section, CafeOBJ maintains a list of search directories. Some of them are *standard*. You may change the list by "-l" or "+l" options, as explained in Section 1.2.3. You may also change it during the session with `set` command (Section 1.3.2). For example,

```
CafeOBJ> set libpath + /usr/home/maria/cafe:/usr/home/tania/cafe
```

adds the two directories to the search list, while

```
CafeOBJ> set libpath /usr/home/maria/cafe:/usr/home/tania/cafe
```

replaces the search list by the twosome. Each pathname should be separated by ":", like in "-l" and "+l" options.

By default, if an apparent single file name is given as argument to `input` command, the system searches the current directory, then standard library directories. The standard library directories are fixed at the time of installation. By default of defaults, they are

```
/usr/local/cafeobj-1.4/lib
/usr/local/cafeobj-1.4/exs
```

which are searched *in this order*. If the search paths are modified, the sytem searches (a) the current directory, (b) the paths given by the user (from left to right), and (c) the above default paths (in case of + or +l options).

Warning The current directory has a privileged status. It is always searched first, and cannot be suppressed by "-l" options or `set` command.

To inspect the current search paths, type

```
CafeOBJ> show switch libpath
```

(see Section 1.3.3 for `show` command in general).

1.2.6 Requiring and Providing

If you are familiar with emacs, you may fancy the following pair of commands for configuration management.

Syntax 9: `require` *command* `require` *feature* [*pathname*]

Syntax 10: `provide` *command* `provide` *feature*

`require` 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.

`provide` discharges a feature requirement: once `provide`d, all the subsequent `require`ments of a feature are assumed to have been fulfilled already. For example, you may be at the point of reading a file that refers to a module of lists. Suppose your convention is to store a set of container modules — lists, bags, sets, and so on — in a single file. Suppose further that, for various reasons, you have a couple of alternative definitions of those modules, stored in files `container-1.mod`, `container-2.mod`, etc. Then it is convenient to insert the command

```
provide container
```

in each file, so that, once one of them read in, the system knows that a module of lists is already supplied.

To see which features have been provided, type

```
CafeOBJ> show features
```

(see Section 1.3.3 for `show` command).

To continue the above example, provided there is a file `container-1.mod` that defines lists, bags, et al. and contains the `provide` command line as above, you may get the following session.

```
CafeOBJ> provide numbers

CafeOBJ> require container container-1

processing input : ./container-1.mod
-- defining module LIST_* done.
-- defining module BAG_* done.
-- defining module SET_* done.
CafeOBJ> require container container-1

CafeOBJ> show features
container numbers
```

The first invocation of `require` made the system read the file, while the second invocation caused nothing. Also, since `numbers` was explicitly `provide`d, the system presumes the feature as given.

1.3 Some Helpful Commands

1.3.1 Help Command and Top-level Commands

Help commands of CafeOBJ in general employ "?", instead of "help", "aide-moi", "socorro", and so on. You may remember that the welcoming message contained the line

```
Type ? for help
```

saying that the topmost help command is

> **Syntax 11:** *command listing command*
> *"?"*

which lists the top-level commands. They may be classified as follows.

- Starting module declarations. `module` (Section 2.1), `view` (Section 8.2.3), and some others introduce building blocks of CafeOBJ codes. They are to be explained in detail, in the rest of the manual.
- Checking properties of modules. `check` (Section 10.1).
- Managing systems. `quit` (Section 1.1), `select` (Section 8.1.1), `set` (Section 1.3.2), `protect` (Section 2.1), and others.
- Managing files. `input` (Section 1.2.1), `save` (Section 1.2.2), and others. They should be already familiar to you.
- Peeping systems. `show` and `describe` (Section 1.3.3) are at your command.
- Evaluating terms. `reduce`, `execute`, and some others. Part of `set` command also concerns evaluation. They are all explained in Chapter 6.
- Helping theorem proving. `apply`, `open`, et al. They shall be explained in Chapter 9.

Many commands have abbreviations. We list them below.

`module`	`mod`	`module!`	`mod!`
`module*`	`mod*`	`quit`	`q`
`show`	`sh`	`describe`	`desc`
`input`	`in`	`reduce`	`red`
`execute`	`exec`	`b-reduce`	`bred`

> **Warning** Other abbreviations may be accepted by the system. But unless explicitly stated in the manual, correct behaviours are not guaranteed.

Some of the commands takes many different kinds of arguments. It is unlikely for you to remember all. In such cases, the system interprets the argument "?" as your desperate call. For example,

```
CafeOBJ> set ?
```

prints a list of arguments acceptable to `set` command, to be introduced immediately.

1.3.2 Switch-Toggling Command

The system maintains various kind of switches. By definition a *switch* is two-valued (`on` and `off`). The following command changes the value of a switch.

> **Syntax 12:** `set` *command for switches*
> `set` *switch_name* { `on` | `off` }

As explained in Section 1.3.1, you get a list of available switches by just typing

```
CafeOBJ> set ?
```

and the current value of switches are listed by

```
CafeOBJ> show switches
```

(for `show` command, see Section 1.3.3).

A switch that affects the general behaviour of the system is `verbose`. After this switch is turned `on`, the system suddenly becomes talkative in responding to many commands, such as `show` and `describe`. Another such switch is `quiet`. If this switch is `on`, the system does not bother you with messages other than blatant error messages. Other switches are relevant to the specific commands of the system, and will be explained as the need arises.

Warning Available switches are subject to frequent changes.

Some arguments to `set` command are not switches, and may range over many values. An example is `libpath`, which was appeared in Section 1.2.5. The general syntax might be

Syntax 13: **`set`** *command for others*
`set` *a_name* [*option*] *value*

but options and values are dependent on the name, and names have no common characteristic other than that they refer to something. We shall explain them individually, as the occasions arise.

1.3.3 Inspection Commands

`show` command prints information on modules, system status, et al. The acceptable arguments are many, so you are wise to remember that

```
CafeOBJ> show ?
```

lists them all (cf. Section 1.3.1).

The general syntax is complicated. It is better understood when divided into two. For the first half, you have

Syntax 14: **`show`** *command for module inspection*
`show` [*module_expression*]
`show` [`all`] { `sorts` | `ops` | `sign` | `axioms` } [*module_expression*]
`show` { `vars` | `params` | `subs` } [*module_expression*]
`show` { `sort` *sort_name* } | { `op` *operator_name* } [*module_expression*]
`show` { `sub` *number* } | { `param` *parameter_name* } | { [`all`] `rules` } [*module_expression*]
`show view` *view_name*

Do not bother to memorise all of these for now — or ever. We illustrate them after module building constructs are explained (cf. Chapter 5). For the second half, you have

Syntax 15: **`show`** *command for other inspection*
`show tree` | { `term` [*let_variable*] [`tree`] } | { `subterm` [`tree`] }
`show` [`let` | `selection` | `pending` | `context` | `modules` | `views` |
`time` | `limit` | `stop` | `features` | `memo`
`show` [`all`] { `rule` *rule_specification* } | { `switches` [*switch_name*] }

Again, do not bother now. Most of these arguments are unrelated, and we explain the system's response separately for each of them, in relevant chapters.

There is a command that enables you to look deeper into module definitions.

Syntax 16: **`describe`** *command*
`describe` [*module_expression*]
`describe` [`all`] { `sorts` | `ops` | `sign` | `axioms` } [*module_expression*]
`describe` { `vars` | `params` | `subs` } [*module_expression*]
`describe` { `sort` *sort_name* } | { `op` *operator_name* } [*module_expression*]

Warning In using the system, you may notice that it treats `show` and `describe` in combination. In fact

```
CafeOBJ> show ?
```

lets you know the syntax of `describe` as well as `show`. Try also

```
CafeOBJ> describe ?
```

Chapter 2

Module Declaration

The basic building blocks of CafeOBJ are modules. This chapter explains the outermost structure of a module.

2.1 Overall Structure

A module without parameters has the following syntax.

> **Syntax 17:** *module without parameters*
> `module` *module_name* "{"
> *module_element* ∗
> "}"

A module name is an arbitrary character string. A module element is either (1) import declaration, (2) sort declaration (with ordering), (3) operator declaration, (4) record declaration, (5) variable declaration, (6) equation declaration, or (7) transition declaration. In the subsequent chapters, we shall introduce each of them in turn.

A general rule about module elements is that a referent must precede in the textual order. For example, you cannot refer to a variable unless you declared it somewhere above. Another general rule is that a module element belongs to the module where it appears, and, unless imported (Section 2.5), it cannot be referred to in other modules.

Different modules with the same name may be declared, but only the latest declaration is maintained. You cannot recover previous declarations. The system gives a warning when a previous declaration is discarded.

Some modules are vital for the system to behave correctly. Those modules cannot be overwritten. There are other special modules, not so vital but worth a diligent protection. An example is a module that defines natural number arithmetic, called `NAT` (cf. Chapter 11). You may remove the guard from those modules, by the command

> **Syntax 18:** `unprotect` *command*
> `unprotect` *module_name*

For the reverse effect, you may use

> **Syntax 19:** `protect` *command*
> `protect` *module_name*

which make it impossible to overwrite the given module. You may use this command to protect your own cherished modules. The following session illustrates the behaviour of the system. In this session, we only use the syntax introduced so far.

```
CafeOBJ> module TEST { }

-- defining module TEST_* done.
CafeOBJ> module TEST { }

-- defining module TEST
[Warning]: redefining module TEST _* done.
CafeOBJ> protect TEST

CafeOBJ> module TEST { }

-- defining module TEST
[Error]: module TEST is protected!
CafeOBJ> unprotect TEST

CafeOBJ> module TEST { }

-- defining module TEST
[Warning]: redefining module TEST _* done.
CafeOBJ>
```

According to the syntax of module declarations, empty modules are allowed, as was the `TEST` here. You are warned even if the unprotected `TEST` is overwritten. Once under protection, `TEST` cannot be redefined. Stripped of protection, it is subject to redefinition again.

2.2 Blocks in a Module

Module elements can be classified in three categories: (1) references to other modules, (2) signature definitions, and (3) axioms. It is possible to gather elements in each category into blocks. References to other modules may be clustered by the following syntax.

> **Syntax 20: `imports`** *block*
> `imports` "{" *module_element* ∗ "}"

where module elements are restricted to import declarations (Section 2.5). Similarly, signature definitions and axioms may be clustered by

> **Syntax 21: `signature`** *block*
> `signature` "{" *module_element* ∗ "}"

> **Syntax 22: `axioms`** *block*
> `axioms` "{" *module_element* ∗ "}"

where module elements in a `signature` block are restricted to declarations of sorts, operators, and records (Chapter 3), while those in an `axioms` block are restricted to declarations of variables, equations, and transitions (Chapter 4).

`imports`, `signature`, and `axioms` blocks may appear more than once in a single module. `signature` may be abbreviated to `sig`, and `axioms` to `axs`.

These block constructs are for the sake of legibility, and have no semantic connotation. For example,

```
module SIMPLE-NAT {
  signature {
    [ Zero NzNat < Nat ]
    op 0  : -> Zero
    op s : Nat -> NzNat
```

```
    op _+_ : Nat Nat -> Nat
  }
  axioms {
    vars N N' : Nat
    eq 0 + N = N .
    eq s(N) + N' = s(N + N') .
  }
}
```

is exactly the same module declaration as

```
module SIMPLE-NAT {
  [ Zero NzNat < Nat ]
  op 0  : -> Zero
  op s : Nat -> NzNat
  op _+_ : Nat Nat -> Nat
  vars N N' : Nat
  eq 0 + N = N .
  eq N + s(N') = s(N + N') .
}
```

or minutely blocked

```
  module SIMPLE-NAT {
  signature {
    [ Zero NzNat < Nat ]
  }
  signature {
    op 0  : -> Zero
    op s : Nat -> NzNat
  }
  signature {
    op _+_ : Nat Nat -> Nat
  }
  axioms {
    vars N N' : Nat
  }
  axioms {
    eq 0 + N = N .
    eq N + s(N') = s(N + N') .
  }
}
```

Warning In inspecting modules by `show` command (Section 1.3.3), you may notice that the system uses these block constructs, even if you did not supply them (cf. Chapter 5).

2.3 Tight Modules, Loose Modules

In CafeOBJ, a module plays two different rôles, depending on where it appears or what it contains. In one guise, a module denotes a unique (up to iso) model (*tight denotation*). In the other, it denotes a class of models (*loose denotation*). The latter is the case when a module is used as parameter (see Section 8.2.1); when it defines behavioural properties (Section 5.2.4), and so on. We do not go into the detail in this manual, but the distinction is very important. You should consult [3] or other technical materials at least once.

Sometimes it is necessary or convenient to restrict the rôle of a module. Alternative module declarations below allow you to do that.

Syntax 23: *module for tight denotation*
`module!` *module_name* "{"
module_element $*$
"}"

Syntax 24: *module for loose denotation*
`module*` *module_name* "{"
module_element $*$
"}"

The syntax inside is the same as declarations that start with `module`.

When a module is declared with "`module!`", it always denotes a unique model. When declared with "`module*`", it always denotes a class of models.

2.4 Comment

A comment of CafeOBJ is one of the followings.

a. From "`**`" to the end of line (eol).

b. From "`**>`" to the eol.

c. From "`--`" to the eol.

d. From "`-->`" to the eol.

Comments may be inserted either between module declarations or between module elements.

A comment that starts with "`**>`" or "`-->`" is displayed when it is input to the system. This facility is useful when a file is being read, or when you want to state the expected results of evaluations. For example, if you are excessively uneasy about what is going on, you may prepare a file that contains

```
**> immediately above TEST
module TEST {
--> just entered into TEST
**> now sort declaration
[ Elt ] -- this comment should be supressed
**> sort declaration ended
--> operator declaration starts
op a : -> Elt -- this comment should be supressed
--> operator declaration ended
**> finishing TEST declarartion
}
--> finished!
```

and feed the file — call it `test` here — to the system. The result is as follows.

```
CafeOBJ> in test

processing input : ./test.mod
**> immediately above TEST
-- defining module TEST
-- just entered into TEST.
```

```
** now sort declaration..

** sort declaration ended.

-- operator declaration starts..

-- operator declaration ended.

** finishing TEST declarartion._* done

--> finished!
CafeOBJ>
```

As shown above, ">" is stripped when the comment appeared within a module declaration.

As an aside, you may have noticed a couple of "."'s in the system messages. These dots indicate that the system is running hard. More dots mean more labour. "_" and "*" just before `done` were also from the system.

2.5 Import Declaration

An import declaration is [sic] to import declarations from another module.

Syntax 25: *import declaration* { `protecting` | `extending` | `using` } "`(`" *module_expression* "`)`"

`protecting` etc. are called *importation modes*, and indicate how to import the designated module. You may abbreviate the modes to `pr`, `ex`, and `us` respectively. A module expression is a designation of a module, and is explained later (Section 8.4). For the current purpose, it is enough to know that an individual module name is by itself a module expression.

To illustrate the meaning of import declarations, here is a contrived example.

```
module BARE-NAT {
  [ Nat ]
  op 0 : -> Nat
  op s : Nat -> Nat
}

module BARE-NAT-AGAIN {
  protecting (BARE-NAT)
}
```

The declaration of `BARE-NAT-AGAIN` is equivalent to

```
module BARE-NAT-AGAIN {
  [ Nat ]
   op 0 : -> Nat
   op s : Nat -> Nat
}
```

i.e. it is a copy, a sad shadow, of `BARE-NAT`. Only the name is changed.

To explain the precise meaning of importation modes, we have to delve into a model-theoretic theatre, which is situated outside of this manual. So let us offer only a cursory glance here. The modes state to what extent the intended models of the imported modules are preserved within those of the importing modules. Intuitively, when `M'` is imported, `protecting` is the strongest commandment. This mode requires all the intended models of `M'` be preserved as they are.

`extending` allows the models of `M'` to be inflated, but does not allow them to be collapsed.

`using` allows total destruction.

Warning To prove model-theoretic properties is very hard, and the system does not check whether indeed these restrictions are observed. It is on the user to choose correct modes.

As a special case, 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. You may prohibit implicit importation by `set` command, as

```
CafeOBJ> set include BOOL off
```

but be careful: Boolean operators are indispensable to write conditional axioms (Section 4.2.2), and an attempt to use them will cause errors if `BOOL` is not imported.

Import declarations define a module hierarchy. We shall probe it later (Chapter 8).

Chapter 3

Signature

A signature decides what is and is not a well-formed term (see Section 3.3.1 for detail). A very basic definition would tell you that it consists of sort (name)s and operator (symbol)s paired with ranks. A signature for CafeOBJ is more elaborate. As briefly stated in the introductory chapter,

- The set of sorts may be partially ordered under inclusion, and
- There are two distinct kinds of sorts, with different concepts of equivalences.

These complications are reflected in the syntactic elements in several ways.

3.1 Sort Declaration

A sort is a set of classified elements, such as a set of designers, of supermodels, and of politicians. CafeOBJ has two kinds of sorts, hereafter called *visible sorts* and *hidden sorts*. We leave it to you to decide whether the sort of politicians is visible or hidden.

3.1.1 Sort Declaration

The simplest sort declaration is as follows.

> **Syntax 26:** *visible sort declaration*
> "`[`" *list_of_sort_names* "`]`"

A sort name is an arbitrary character sequence, and a list of sort names is a list of such names separated by blanks. For example,

```
[ Nat ]
```

declares a single visible sort `Nat`, while

```
[ Nat Int Rat ]
```

declares visible sorts `Nat`, `Int`, and `Rat`. By replacing "`[`" and "`]`" with "`*[`" and "`]*`" respectively, you get a hidden sort declaration.

> **Syntax 27:** *hidden sort declaration*
> "`*[`" *list_of_sort_names* "`]*`"

In declaring operators (Section 3.2.1) and axioms (Chapter 4), you have to distinguish visible sorts and hidden sorts. They give rise to different kinds of models, and are totally disjoint. Syntactically, you must not use the same name for both a visible and a hidden sort.

Warning The current implementation simply ignores dual declarations. For example, in the presence of a hidden sort `S`, the declaration

```
[ S ]
```

is silently ignored.

3.1.2 Subsort Declaration

You may define a partial order over sets of sorts, which is interpreted as set inclusion. Such an order is denoted with "<". If `A < B`, we say `A` is a *subsort* of `B`, and `B` a *supersort* of `A`. To declare this relation, you use

Syntax 28: *subsort declaration*
"`[`" *list_of_sort_names* "`<`" *list_of_sort_names* { "`<`" *list_of_sort_names* } $*$ "`]`"
"`*[`" *list_of_sort_names* "`<`" *list_of_sort_names* { "`<`" *list_of_sort_names* } $*$ "`]*`"

"`[`", "`]`" enclose visible sorts and "`*[`", "`]*`" hidden ones, as before. A list of sort names is as in a sort declaration, but it can be qualified (qualification is explained in Section 8.1.3). The above syntax means that each sort in the first list is a subsort of each sort in the second, each sort in the second is a subsort of each sort in the third, and so on. Note that "<", although looking a strict relation, is non-strict, and means "less than or equal to", or "subset of or equal to".

For example, the usual inclusion of natural numbers `Nat` into integers `Int`, and `Int` into rationals `Rat`, may be written down as

```
[ Nat < Int < Rat ]
```

or

```
[ Int < Rat ]
[ Nat < Int ]
```

or

```
[ Nat Int < Rat ]
[ Nat < Int ]
```

For a briefer description, you may put more than one declarations within a single bracket pair, using commas for separation. For example, the last declarations above may be abbreviated to

```
[ Nat Int < Rat, Nat < Int ]
```

It is also possible to combine sort and subsort declarations into a single bracket pair, as

```
[ Nat Int Rat, Nat < Int < Rat ]
```

which is equivalent to

```
[ Nat Int Rat ]
[ Nat < Int < Rat ]
```

Warning In subsort declarations, if new sort names appear, they are assumed to be declared implicitly. If `Nat`, `Int` and `Rat` have not been declared, therefore,

```
[ Nat < Int < Rat ]
```

and

```
[ Nat Int Rat,  Nat < Int < Rat ]
```

are interpreted identically. Beware, especially, of mistyping upper/lower cases. For example, if you imported a sort `Nat` and declared

```
[ NAT < Int ]
```

`NAT` is regarded as a newly declared sort, not an imported one.

It is abnormal for subsort relation to be circular, and if they were circular, you should not expect the system to behave correctly. Subsort relation is set inclusion, so circular relation means that all the sorts involved are the same. But CafeOBJ does not recomment such indirect "renaming", and may refuse to support this interpretation.

Warning In general, checks on circularity are hard to implement efficiently, so the system only checks obvious cases. If circularity is detected, you are severely warned.

Note that it is impossible to make a visible sort a subsort of a hidden sort, or vice versa. You cannot put a hidden sort in plain brackets, nor a visible sort in asterisked brackets. This impossibility reflects the fact that visible sorts are disjoint from hidden sorts.

3.2 Operator Declaration

In traditional algebraic specification languages, an operator is a function, i.e. an unambiguous, static mapping. CafeOBJ extends the concept in two directions. An operator may *transform* an element into another. Such an operator is defined by transition relations (Section 4.3). Moreover, an operator may change an *internal state* of an element. Such an operator is defined from an observational viewpoint.

3.2.1 Operator Declaration

Like the other strongly-typed languages, CafeOBJ requires you to make explicit the domains and codomains of operators, and only allows terms constructed accordingly. Since CafeOBJ is order-sorted, however, well-formedness involves subtle considerations (see Section 3.3.1).

In addition to domains and codomains, operator declarations determine term constructs. CafeOBJ allows you to employ various notations in writing terms, such as infix (e.g. $2+4$), prefix (-4), and postfix ($2!$). Hence in declaring operators, you have a way to adopt your favourite notational practice. If you do not show any preference, you get a standard notation for function application. For example, given a (ternary) operator `f`, a term thereof is constructed out of parenthesised, comma separated arguments, such as `f(a,b,c)`. A declaration for such an operator is called *standard.*

Syntax 29: *standard operator declaration*
`op` *standard_operator_symbol* "`:`" *list_of_sort_names* "`->`" *sort_name*

A standard operator symbol is an arbitrary character string that does not contain an underbar "_". A list of sort names is a blank-separated list and designates the arity of the operator. The sort name after "->" designates its coarity. Sort names in arities and coarities may be qualified (cf. Section 8.1.3). The pair of the arity and the coarity is called the rank of the operator. An operator with the empty arity is a constant.

Warning It is possible for a standard operator symbol to contain blanks. For example, in

```
op a constant of sort S : -> S
```

the whole "`a constant of sort S`" is a symbol. More precisely, the system treats the symbol as a blank-separated list of non-blank character strings, and arbitrary numbers of blanks may be inserted. But parsing terms that contain such symbols tends to be erroneous, and you often have to enclose the symbol with parentheses. For this reason, it is advisable not to use blanks, unless your aesthetics so forces.

If not standard, a declaration is called *mixfix*.

Syntax 30: *mixfix operator declaration*
op *mixfix_operator_symbol* ":" *list_of_sort_names* "->" *sort_name*

A mixfix operator symbol may contain underbars. Arities and coarities are the same as in standard operator declarations. In fact, the only difference from standard declarations is the presence of underbars. Underbars reserve the places where arguments are inserted. The leftmost underbar is for the first argument, the next leftmost the second, and so on. The number of underbars must match the number of sort names in the arity.

Warning The single underbar cannot be an operator symbol; it would introduce invisible operator applications. Such a declaration is ignored, and you are ridiculed by the system.

For example, addition is usually written in the infix notation. Declare

```
op _+_ : Nat Nat -> Nat
```

and terms such as "`N + M`", "`s(0) + N`" are admissible. Another representative example is a conditional expression.

```
op if_then_else_fi : Bool Nat Nat -> Nat
```

According to the definition, this declaration allows you to write

```
if N < M then N else M fi
```

and

```
if true then 0 else s(0) fi
```

Since CafeOBJ allows operator overloading, it is possible to use the same operator symbol for different operators. For example, the symbol `_+_` may be overloaded as follows.

```
op _+_ : Nat Nat -> Nat
op _+_ : Set Set -> Set
```

An operator symbol, whether standard or mixfix, does not only define term constructs, but acts as an operator name. What an operator name refers to is a nontrivial question, due to overloading possibilities (see Section 8.1.4 for detail).

You can declare several operators simultaneously, if they have exactly the same rank: use ops instead of op, as

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

which means the same as

```
op _+_ : Nat Nat -> Nat
op _*_ : Nat Nat -> Nat
```

Note that, in simultaneous declarations, parentheses are sometimes necessary to separate operator symbols.

Warning A general advice: if you are in doubt, enclose with parentheses. Parentheses help you avoid many parsing ambiguities.

3.2.2 Behavioural Operator Declaration

For hidden sorts, you need a special kind of operators, called *behavioural operators*.

Syntax 31: *behavioural operator declaration*
`bop` *standard_operator_symbol* ":" *list_of_sort_names* "->" *sort_name*
`bop` *mixfix_operator_symbol* ":" *list_of_sort_names* "->" *sort_name*

Standard operator symbols, mixfix operator symbols, lists of sort names and sort names are as in `op` in the previous section. It is required that

- The arity of a behavioural operator contain exactly one hidden sort.

Intuitively, you may regard a hidden sort as consisting of objects, which have internal states. Then a behavioural operator is an *attribute* or a *method*, if you really, really want to adopt the parlance of a certain community. As an illustration, consider bank account objects, whose balances change upon deposits and withdrawals.

```
*[ Account ]*
bop deposit : Account Nat -> Account
bop withdraw : Account Nat -> Account
bop balance : Account -> Nat
```

Here `deposit`, `withdraw` are operations that modify internal states — balances — of accounts (so they are methods), while `balance` reads the current balances of accounts (so an attribute).

The above intuition tells you why a behavioural operator should have exactly one hidden sort in its arity. It does not make sense to retrieve attributes from two objects. And it is nonsense to talk about attributes of nothing. Technically, behavioural operators, and only them, make behavioural contexts (see Section 5.2.4).

There is a counterpart to `ops`, with which you may declare several behavioural operators at once. Two of the above declarations may be replaced by

```
bops deposit withdraw : Account Nat -> Account
```

Note that an operator symbol may be overloaded among behavioural operators, and/or among the mixture of usual and behavioural operators. Note further that a non-behavioural operator declaration with `op` is *not* required to contain or not to contain hidden sorts. You may declare non-behavioural operators on hidden sorts, as

```
op deposit : Account Nat -> Account
```

The effects are, however, very different. Do take care.

3.2.3 Predicate Declaration

A *predicate*, in CafeOBJ, is nothing more than an operator which has a special sort `Bool` as its coarity. This sort is contained in the module `BOOL`, which is imported everywhere (Section 2.5). `Bool` is important in several respects, and is accorded a privilege by the following shorthand constructs.

Syntax 32: *predicate declaration*
  `pred` *standard_operator_symbol* " : " *list_of_sort_names*
  `pred` *mixfix_operator_symbol* " : " *list_of_sort_names*

Operator symbols etc. are as in operator declarations. Predicate declarations are equivalent to operator declarations with coarity `Bool`. For example,

```
pred _<_ : Nat Nat
```

has the same meaning as

```
op _<_ : Nat Nat -> Bool
```

Warning The system does not distinguish predicates as such. You should think of them as mere shorthands. Nothing more.

To complete the picture, `bpred` is a shorthand of `bop` with coarity `Bool`.

Syntax 33: *behavioural predicate declaration*
  `bpred` *standard_operator_symbol* " : " *list_of_sort_names*
  `bpred` *mixfix_operator_symbol* " : " *list_of_sort_names*

3.3 Terms

3.3.1 Well-formed Terms

As well as adopting arbitrary notational practices, you may use the same notation to denote different operators, such as $1+2$ for addition and $\{a\}+\{a,b\}$ for direct sum. These flexibilities greatly enhance the legibility of CafeOBJ codes, and we believe it worthwhile to support them.

One disadvantage of this liberty is the extra amount of processing time in parsing terms. Another, and more important, problem is that a term may be parsed in more than one ways. We make recourse to the following definition.

> A term is *well-formed* iff it has exactly one least parse tree.

Several parse trees are okay, so long as all of them are of related sorts, and one of them has a *least sort*. The least sort is the smallest set to which the element belongs. Dual membership is a seed of troubles, just as dual marriage is.

A signature is *regular* iff all the terms have least sorts, and a module is regular iff its signature is. With the above observations in mind, you understand this is an important property. CafeOBJ has a mechanism to regularise signatures. To invoke this mechanism, you may use

Syntax 34: **`regularize`** *command*
  `regularize` *module_name*

or set a switch for automatic invocation to on by

```
CafeOBJ> set regularize signature on
```

The default is `off`. To regularise a signature, additional sorts and operator declarations may be declared. These addidions ensure the existence of "lowest" operators. For a trivial example, in

```
module INTB {
  [ NonPos Nat < Int ]
  op 0 : -> NonPos
  op 0 : -> Nat
}
```

the constant 0 belongs both to NonPos and to Nat. In general it is possible to overload constants, there is a way to distinguish them (see Section 3.3.2), and there is no harm in that. But in this example, since NonPos and Nat are both subsorts of Int, the constant must in fact denote the same element. As a consequence, the term 0 does not have a least parse (even qualified, as in Section 3.3.2). A remedy is to add a common subsort of NonPos and Nat, and declare 0 of that sort. Using `regularize` command, you do get a module modified along this line:

```
module INTB {
  [ NonPos^Nat < NonPos Nat < Int ]
  op 0 : -> NonPos^Nat
}
```

Two old declarations of 0 were removed since they were redundant by now.

Warning The current version does not remove redundant 0's.

More generally, the system searches for operators that may create terms with no least parse, and declare such operators on least sorts after, if necessary, adding new common subsorts. What the system checks is a sufficient, but *not* necessary, condition. There are cases the system is overprotective. As an example, consider

```
module INTB' {
  [ NonPos Nat < Int ]
  op _+_ : NonPos NonPos -> NonPos
  op _+_ : Nat Nat -> Nat
  op _+_ : Int Int -> Int
  op 1 : -> Nat
  op -1 : -> NonPos
}
```

INTB' is regular: a term is of the form of a binary tree where all the nodes contain + and all the leaves contain 1 or -1, and is of sort NonPos iff every leaf is -1, of sort Nat iff every leaf is 1, and of sort Int iff otherwise. If you invoke `regularize` command, however, you get a modified module:

```
module INTB' {
  [ NonPos^Nat < NonPos Nat < Int ]
  op _+_ : NonPos^Nat NonPos^Nat -> NonPos^Nat
  op _+_ : NonPos NonPos -> NonPos
  op _+_ : Nat Nat -> Nat
  op _+_ : Int Int -> Int
  op 1 : -> Nat
  op -1 : -> NonPos
}
```

The reason is as follows. If Nat, NonPos are both inhabited, there is a danger that their intersection is not empty. If it is indeed non-empty, an application of "_+_" on any terms in it creates a term both of sort Nat and of NonPos. So it is safe to add a new common subsort, and declare "_+_" on it.

To check the non-emptiness is too hard to solve mechanically. For example, if you insert an equation (Section 4.2.1)

```
eq 1 = -1 .
```

into `INTB'`, there does exist an element, namely `1` (or `-1`), which belongs to both `NonPos` and `Nat`. Thus a general solution requires a complete equational calculus, which is beyond the power of CafeOBJ. It is for this reason that the system always bets on the safer side, even if the results are dull: the system (and you) will not lose a penny, since the modified module has exactly the same model(s) as before (when you take reducts).

If one of `NonPos` or `Nat` were empty, the system would have done nothing. For example, if you delete the declaration

```
op 1 : -> Nat
```

from `INTB'`, `Nat` is not inhabited any more: the system adds nothing this time.

For those who only want to be warned of the danger, it is also possible just to check if the signature is regular (cf. Section 10.1).

3.3.2 Parsing Terms

As suggested above, a symbol sequence may be parsed in more than one ways. A salient example is a constant symbol. With the two declarations below, `empty` is a term both of `Stack` and `Set`.

```
op empty : -> Stack
op empty : -> Set
```

In such cases, you have to supply additional information by qualification. Other ambiguities may be eliminated by parentheses. Let us see a couple of examples.

To see whether a term is unambiguous, the following command is handy.

Syntax 35: *parsing command*
parse [**in** *module_expression* ":"] *term* " " "."

If **in** option is omitted, the current module (cf. Section 8.1.1) is assumed. Module expressions are explained in Section 8.4. It is enough here to know that a module name is a module expression.

Warning Do not forget the last period, preceded by a blank. This is the only way for the system to detect the end of a term: usual parsing techniques would not do, due to flexible term constructs. The same convention is used in other commands and declarations.

To see how this command works, here is an example where an ambigouous term is parsed.

```
module SIMPLE-INT {
  protecting (SIMPLE-NAT)
  [ Nat NegInt < Int ]
  op -_ : NzNat -> NegInt
  op _+_ : Int Int -> Int
  op _-_ : Int Int -> Int
  op s : Int -> Int
  vars M M' : Int
}
```

The above module gives a signature for integer operations (`SIMPLE-NAT` was in Section 2.2). **vars** declares variables (Section 4.1). After declaring the above module, you may get a session like

```
CafeOBJ> parse in SIMPLE-INT : M - M' - s(0) .

[Warning]: Ambiguous term:* Please select a term from the followings:
[1] _-_ : Int Int -> Int ------------------------------------------
          _-_:Int
      /             \
   _-_:Int       s:NzNat
  /       \         |
M:Int  M':Int    0:Zero

[2] _-_ : Int Int -> Int ------------------------------------------
     _-_:Int
  /           \
M:Int       _-_:Int
          /         \
       M':Int    s:NzNat
                    |
                 0:Zero

* Please input your choice (a number from 1 to 2, default is 1)? 2
Taking the second as correct.

(M - (M' - s(0))) : Int
CafeOBJ>
```

If the term is ambiguous, as in this example, the system gives a warning, and threatens you to be more resolute. You must choose one definite parse. The system numbers possible choices and displays the parse tree for each choice. The number after the message "`* Please ...`" is your input.

> **Warning** The system is impatient. If you hesitate long, it automatically selects the default (in the above case, 1).

The result of the parsing was

```
(M - (M' - s(0))) : Int
```

where the term is parenthesised to reveal connectivity. `Int` on the right of ":" is its least sort.

3.3.3 Qualifying Terms

Ambiguity that arises from operator connectivity, e.g.

```
M - M' - s(0)
```

above, can be eliminated by parentheses. On the other hand, ambiguity caused by overloading cannot be so easily eliminated. It requires additional sort information. Hence term qualification, as

> **Syntax** 36: *sort-qualified term*
> "(" *term* ")" ":" *sort_name*

> **Warning** This version, unlike former versions, allows blanks around ":". The following terms are all valid and parsed as the same.
>
> ```
> (empty):Stack
> (empty): Stack
> (empty) :Stack
> (empty) : Stack
> ```

Warning Be sure to put parentheses around the qualified. CafeOBJ allows inline variable declarations, such as

```
M:Nat - s(0)
```

where `M` is regarded (at this place only) as a variable of sort `Nat`. Without parentheses, e.g.

```
empty:Stack
```

is regarded as a new variable!

For example, consider the following module declarations.

```
module OCTAL {
  [ OctalDigit ]
  ops 0 1 2 3 4 5 6 7 : -> OctalDigit
}

module HEX {
  [ HexDigit ]
  ops 0 1 2 3 4 5 6 7 8 9 : -> HexDigit
  ops A B C D E F : -> HexDigit
}

module HEX+OCTAL {
  protecting (OCTAL)
  protecting (HEX)
}
```

Since `HEX+OCTAL` contains declarations either in `OCTAL` or in `HEX`, the digits `0` through `7` are ambiguous there: they are constants both of `OctalDigit` and `HexDigit`. Indeed, try parsing `0` and you get a warning.

```
CafeOBJ> parse in HEX+OCTAL : 0 .

[Warning]: Ambiguous term:* Please select a term from the followings:
[1] 0 :  -> OctalDigit ----------------------------------------------
0:OctalDigit

[2] 0 :  -> HexDigit ------------------------------------------------
0:HexDigit

* Please input your choice (a number from 1 to 2, default is 1)? 1
Taking the first as correct.

0 : OctalDigit
CafeOBJ>
```

To eliminate this ambiguity, you may qualify by sorts. Try again with

```
CafeOBJ> parse in HEX+OCTAL : (0):OctalDigit .
```

and the system will oblige with

```
0 : OctalDigit
```

3.4 Record Declaration

A *record* declaration defines a record-type data structure. A record type consists of fields, and an element of the type is completely determined by the value of each field. In CafeOBJ, a field is called *slot*.

3.4.1 Plain Record Declaration

A record declaration without inheritance (see the next section) is as follows.

Syntax 37: *plain record declaration*
`record` *record_name* "{" *list_of_slot_declarations* "}"

where a list of slot declarations is a blank-separated list. A slot declaration has the following construct.

Syntax 38: *slot declaration*
slot_name ":" *sort_name*

where a slot name is a character string and a sort name is the name of an already declared sort. For example, if `Nat` is a sort of natural numbers with usual digit notations,

```
record Date {
  year  : Nat
  month : Nat
  day   : Nat
}
```

is a sloppy definition of records that consist of years, months, and days. (Sloppy because it admits e.g. 554/32/1984 as a date.)

In ordinary programming languages, once a record is declared, notation/functions to access fields are automatically determined. The dotted notation `d.day`, where `d` is a record variable and `day` is a field name, is typical. CafeOBJ also provides access functions. In fact, a record is nothing more than a special sort. The effects of a record declaration are as follows.

- A sort with the record name is declared. A record is a sort, so the name can appear wherever a sort name can.
- A mixfix operator is declared, so that a term of the form

  ```
  record-name { slot-name1 = value1, slot-name2 = value2, ... }
  ```

  is an element of `record-name`. Using the above example,

  ```
  Date { year = 1984, month = 1, day = 1 }
  ```

  is a term of sort `Date`. Moreover, the slot-value pairs in the braces may be written in whatever order, and you may even omit some of them. Thus

  ```
  Date { month = 1, day = 1, year = 1984 }
  Date { year = 1984 }
  ```

  are well-formed terms of sort `Date`. Note that `Date` in these terms is part of an operator symbol, not the sort name which happens to be the same character string.

- For each slot, two access functions are declared and defined. In case of `Date`, it is as if the declarations

  ```
  op year : Date -> Nat
  op set-year : Date Nat -> Date
  -- similarly for month and day
  ```

  are given. Accordingly, to get the value of `year` slot from a term `d` of sort `Date`, you can write `year(d)`; and to set the slot to y, write `set-year(d,y)`.

Strictly speaking, since the definitions of access functions are by axioms (axioms are to be explained in Chapter 4), a record declaration involve axioms as well as signatures. You may investigate all the above effects by `show` command (cf. Section 5.1.4).

3.4.2 Inheriting Records

You can extend a record to create a new record. The same access functions can be used as before. The process of extending a record is called *record inheritance*, à la peculiar parlance of some community, and is denoted by the following construct.

Syntax 39: *record declaration with inheritance*
`record` *record_name* "`[`" *list_of_upper_records* "`]`" "`{`" *list_of_slot_declarations* "`}`"

A list of slot declarations is as in a plain record declaration. A list of upper records is comma-separated. An upper record is of the form

Syntax 40: *upper record*
record_name ["`*`" "`(`" *slot_mapping* "`)`"]

where a slot mapping is

Syntax 41: *slot mapping*
slot_name "`->`" *slot_name* { "`,`" *slot_name* "`->`" *slot_name* } ∗

where slot names before "`->`" must be the ones belonging to the given record, while names after "`->`" are arbitrary (so long as not conflicting).

Under a record declaration with inheritance, the newly declared record inherits all the slots from the upper records. Slot mappings, which specify renaming, are to be used to avoid name conflicts. CafeOBJ does not allow slots of the same name to be inherited twice. If there is no conflict, slot mappings may be omitted.

A boring example of inheritance, using `Date` of the previous section, is

```
record Time [ Date ] {
  hour   : Nat
  minute : Nat
}
```

This code declares `Time` to have slots `year`, `month`, and `day`, as well as `hour` and `minute`. Access functions `year`, `set-year`, and so on are applicable to terms of `Time` exactly as they are to those of `Date`. The effects of the declaration are equivalent to

```
record Time {
  year   : Nat
  month  : Nat
  day    : Nat
```

```
  hour   : Nat
  minute : Nat
}
```

Here is another example, which needs slot renaming.

```
record Week {
  day : Nat
}

record Time'
 [ Date * ( day -> day-of-the-month ), Week * ( day -> day-of-the-week ) ]
{
   hour   : Nat
   minute : Nat
}
```

Since `day` was the slot name of both `Date` and `Week`, each slot was accorded a more accurate, distinct name.

Chapter 4

Axioms

CafeOBJ inherits the essences of algebraic specification techniques, where axioms are given as equations. Several extensions, however, lead to the following complications. (1) Equations may be conditional. (2) Not only equations, but transitions are available. (3) Visible sorts and hidden sorts demand different kinds of equivalences, so that, as well as ordinary equations, behavioural equations are available. All of these extensions are orthogonal, as witnessed by $8 = 2^3$ syntatical constructs.

4.1 Variable Declaration

Axioms require representations of classes of elements, as well as specific elements. You have to be able to say, e.g., that the successor of *any* natural number is greater than that number. A *variable* represents this "any". In CafeOBJ, every variable belongs to a sort, and represents an arbitrary term of that sort. Variables are declared as follows.

Syntax 42: *variable declaration*
`var` *variable_name* "`:`" *sort_name*
`vars` *list_of_variable_names* "`:`" *sort_name*

A variable name is a character string, and a list of variable names is blank-separated. Here is an example.

```
var N : Nat
vars A B C : Nat
```

Warning In former versions, `var` and `vars` are interchangeable. This version becomes linguistically more strict. `var` is for exactly one variable, while `vars` is for one or more variables.

Warning In former versions, commas could be used as separators, so that the above declaration may be replaced with

```
vars A, B, C : Nat
```

But this construct no longer works: the system would regard `,` as another variable.

A simple way to know the typing discipline of a language is to see variable declarations. CafeOBJ adopts a *strong typing* discipline, in that each variable has a syntactically determined type (sort). Another point of view comes from the definition of well-formedness. A term is well-formed iff it has a unique least parse (Section 3.3.1). A variable is a term. Hence it must have a unique least sort, the sort given in the declaration.

The scope of a variable is within the module where it is declared, or more precisely, in the rest of the module after the declaration. For example,

```
module M {
  [ S ]
  op f : S -> S
  -- axioms using X -- a.
  var X : S
  -- axioms using X -- b.
}
```

would be an error, if you put axioms in `a.` involving `X`. The correct place for such axioms is `b.`

It is illegal to declare a variable of the same name for different sorts. (And it is meaningless to re-declare the variable for the same sort.) Sometimes it is b.c.b.g.[1] to use common variable names, such as `X`, for different sorts. And sometimes it is economical to use an ephemeral variable, valid only to the place where it appears. For these reasons, another construct for variable declaration is available.

> **Syntax 43:** *variable declaration on the fly*
> *variable_name* "`:`" *sort_name*

> **Warning** Do not insert blanks on either side of the colon.
>
> ```
> N: Nat
> ```
>
> or
>
> ```
> N :Nat
> ```
>
> is unacceptable. Only `N:Nat` will do.

The following session illustrates the utility of declarations of this form. Recall from Section 3.3.2 a module called `SIMPLE-INT`.

```
CafeOBJ> parse in SIMPLE-INT : X:Int - s(0) .

(X - s(0)) : Int
CafeOBJ>
```

The variable `X` was declared on the fly. If you only want to check syntactic ambiguity, it is a bother to declare variables separately, just for that purpose. The system treats an ephemeral variable declaration as a term — nothing more, nothing less —, so you can declare on the fly wherever you can write a term. For the exact scope of on-the-fly declarations, see the next section.

4.2 Equation Declaration

4.2.1 Unconditional Equation Declaration

A plain equation is an axiom. Anticipating conditional cases, we call it *unconditional*.

> **Syntax 44:** *unconditional equation declaration*
> `eq` [*label*] *term* "=" *term* "`.`"

The last period, preceded by a blank, is a must. It is the way to notify the system of the end of a term. "=" separates the lefthand side from the righthand.

[1] Bon chic, bon genre.

Warning "=" in equations is used by the system to detect the end (and start) of a term. You should supply blanks on both ends, as in

```
eq X + 0 = X .
```

Without blanks, the system thinks the lefthand side is continuing. For example,

```
eq X + 0= X .
```

causes the system to treat the entire text after **eq** (and the subsequent text, until it encounters a lonely "=") as the term on the lefthand side.

Since an equation is a statement that the two sides are equal, they must belong to a common universe of discourse. For example, to say that Marie Antoinette equals 3.1416 is nonsense[2]. In many-sorted logic, this consideration is formalised as "two terms must be of the same sort". CafeOBJ, which is based on order-sorted logic, is more accommodating:

> The sorts of the terms must belong to the same connected component (in the graph defined by the sort ordering).

Accordingly, 1 may be equal to 1.0, to 2/2, to $\sqrt{1}$, and to $0i + 1$ (under the usual number hierarchy). We also require that

> The terms be of visible sorts.

For terms of hidden sorts, behavioural equations (Section 4.2.3) should be used.

Labels do not affect the semantics, and may be omitted. If not omitted, it must be of the form

Syntax 45: *label in axioms*
"`[`" *label_name* "`]`" "`:`"

where a label name is a character string without blanks.

Since most non-trivial equations involve variables, and since unambiguous parsing is a non-trivial question, in examples below we put variable declarations near equation declarations.

To state that `0` is a right identity of addition, you may write

```
var N : Nat
eq [ r-id ] : N + 0 = N .
```

or, using an on-the-fly declaration,

```
eq [ r-id ] : N:Nat + 0 = N .
```

Look at the second occurrence of `N` in this equation. In general, the scope of an on-the-fly declaration is the whole enclosing term (or more precisely, in the rest of the term). For example, the second `N` of

```
N:Nat + s(N)
```

is the same `N` as declared on the fly, since they are both in the same term. As a special rule, a variable declared within an axiom is valid throughout that axiom. The above equation illustrates this special rule.

On-the-fly declarations may overwrite the previous declarations. For example, in

[2] Or are they both pie?

```
module S {
  [ S T ]
  op f : S -> S
  var X : T
  eq f(f(X:S)) = f(X) .
}
```

the variable `X` was declared as of sort `T`, but re-declared on the fly as of sort `S`. Remember that overwriting variable declarations *terra firma* are illegal (Section 4.1). On-the-fly declarations are not subject to this prohibition. Moreover, on-the-fly declarations may themselves be overwritten. The following example uses `X` first as of sort `S`, then as of `T`.

```
module S {
  [ S T ]
  op f : S T -> T
  eq f(X:S,X:T) = X .
}
```

Warning In spite of the above remark, it is advisable not to overwrite on-the-fly's by on-the-fly's. It would easily lead to confusion. Accordingly, the system gives a warning when it encounters such cases.

4.2.2 Conditional Equation Declaration

An equation may be *conditional*. A conditional equation is of the form

Syntax 46: *conditional equation declaration*
`ceq` [*label*] *term* "=" *term* `if` *boolean_term* "."

Labels and terms are as in `eq`. As stated often enough, the last blank-period lets the system know the end of a term, and is a must. Note that a declaration starts with `ceq`, not `eq`. `ceq` may be abbreviated to `cq`.

Warning Like the last period, "=" and `if` demarcate terms. You should put blanks around them.

Warning In former versions, conditions were preceded by `:if`, not `if`.

A boolean term is a term of sort `Bool`, which is declared in a built-in module called `BOOL` (to be more precise, in a built-in module imported by `BOOL`). A conditional equation states that the two terms are equal if the boolean term is true (i.e. it equals the constant `true`). We may say, therefore, that a boolean term *is* a condition. As mentioned earlier (Section 2.5), every module implicitly imports the module `BOOL`, unless you have locked it out. A major reason is that `BOOL` provides you with boolean terms. `BOOL` contains all the usual boolean operators, so that, as conditions, the following terms may be used (if `B`, `B'` are boolean terms).

```
true
false
not B
B and B'
B or B'
```

A complete listing may be obtained by `show` command (see Section 5).

4.2.3 Behavioural Equation Declaration

To state behavioural equality explicitly, the following constructs are available.

Syntax 47: *behavioural equation declaration*
`beq` [*label*] *term* "=" *term* "."
`bceq` [*label*] *term* "=" *term* `if` *boolean_term* "."

`bceq` may be abbreviated to `bcq`. Labels etc. are as in `eq` or `ceq`. Also as in `eq` or `ceq`, both terms must belong to the same connected component.

Warning According to the above definition, you can write behavioural equations over visible sort. There is no harm in that: such equations are treated just like equations with `eq` or `ceq`. But do avoid using `beq` if `eq` is what you mean. Legality does not imply good manners.

Note that in many cases, behavioural equality is defined, indirectly, with ordinary equations (declared with `eq` or `ceq`), not with behavioural ones (cf. Section 5.2.3). Behavioural equations are reserved for cases where explicit statements are desirable: typically, when you state theorems, or want to define behavioural operators in terms of others.

4.3 Transition Declaration

Equality is an equivalence relation, obeying three basic properties of binary relations: reflexivity, symmetry, and transitivity. Dropping one of these properties engenders interesting relations. Partial equivalence is one of them. By dropping symmetry, we get a *transition* relation.

The following constructs define such a transition relation. As with equations, there are conditional and unconditional transitions, and ordinary and behavioural ones. Crossing these, you get

Syntax 48: *transition declaration*
`trans` [*label*] *term* "=>" *term* "."
`ctrans` [*label*] *term* "=>" *term* `if` *boolean_term* "."
`btrans` [*label*] *term* "=>" *term* "."
`bctrans` [*label*] *term* "=>" *term* `if` *boolean_term* "."

Syntactically, transitions are the same as equations, if you replace the equality sign by the arrow "=>". Similar restrictions apply here also. For example, the sorts of two terms must be in the same connected component. `trans` may be abbreviated to `trns`, `ctrans` to `ctrns`, `btrans` to `btrns`, and `bctrans` to `bctrns`.

The following little example illustrates the difference between equations and transitions.

```
module CHOICE {
  [ State ]
  ops a b : -> State
  op _|_  : State State -> State
  vars X Y : State
  trans X | Y => X .
  trans X | Y => Y .
}
```

"`_|_`" denotes an indeterminate state, turning out to be either argument (which itself may be indeterminate). The above rules say that an indeterminate state may become more and more determinate, but not conversely. It is obvious that, if these rules were equations, all the states would be the same, so that `State` would be an uninteresting singleton.

A traditional way to describe the above choice operator is to define an explicit transition function. The technique works at times, but in this nondeterministic example, it would lead to something like

```
module CHOICE {
  [ State ]
  ops a b : -> State
```

```
    op _|_  : State State -> State
    op choose : State -> State
    vars X Y : State
    eq choose(X | Y) = X .
    eq choose(X | Y) = Y .
    eq choose(a) = a .
    eq choose(b) = b .
  }
```

which, alas, is a specification of singletons.

4.4 Internalising Axioms

Axioms define certain kinds of relations when we construct models. For example, axioms introduced by `eq` or `ceq` define minimal congruence relations between reachable (term-generated) elements of visible sorts. These relations reside outside of CafeOBJ descriptions: the mark "=" is just a notation.

For various purposes, it is convenient if these relations can be manipulated within the syntactical framework of CafeOBJ. In general, it is impossible to give general decision procedures for these relations, but under certain conditions, or in some restricted sense, it is possible to implement procedures that realise or surrogate these relations. CafeOBJ provides four predicates for this purpose.

- == for equality.
- ==> for transition relations.
- =*=, =b= for behavioural equivalence.

The latter three are mainly used in theorem proving (cf. Section 10.2).

In addition, the system supports a sort predicate "`:is`".

4.4.1 Equality Predicate

The predicate for (ordinary) equality, written "==", is a binary operator defined on each visible sort. An explicit declaration of the operator would be of the form

```
  op _==_ : S S -> Bool
```

or

```
  pred _==_ : S S
```

for each sort `S`. "_==_" is defined in terms of evaluation (Chapter 6). For ground terms `t`, `t'` of sort `S`, "`t == t'`" equals `true` iff `t`, `t'` evaluate to a common term. It takes care of equational theories, such as associativity (Section 7.1). Due to this definition, "_==_" works correctly only if the term rewriting system is terminating and confluent (Section 6.1.2).

The converse of "_==_" is "_=/=_", which returns `true` iff "_==_" returns `false` on the same arguments.

The exact mechanism for defining these operators is tricky, but all are declared and defined in a built-in module `BOOL`. This module contains usual boolean operators, such as conjunction, and, as mentioned earlier, is implicitly imported by other modules (Section 2.5).

Note the difference between equations, such as

```
  eq N + 0 = N .
```

and applications of the *operator* "_==_", such as

```
N + 0 == N
```

The sign "=" in the former *denote*s an equivalence relation, and this relation per se steer wide of syntactic manipulation. "_==_" is an operator that generates terms, which are subject to manipulation within the language processors. Still, they are related as follows: if the underlying term rewriting system is confluent and terminating,

`X = X'` holds iff `X == X' = true` does.

4.4.2 Transition Predicate

The predicate for transition relation, written "==>", is a binary operator defined on each visible sort. As with "==" in the previous section, an explicit declaration would be of the form

```
pred _==>_ : S S
```

for each sort `S`. This predicate is in fact declared on the "universal" sort, in a module called `RWL`. `RWL` is imported implicitly, when a module contains transition declarations (Section 4.3). As with `BOOL` (Section 2.5), it is possible to switch off this implicit importation, by the command

```
CafeOBJ> set include RWL off
```

A transition relation is reflexive, transitive, and is closed under operator application. When `RWL` is imported, the system supplies a set of axioms. Firstly, for each visible sort `S`, an axiom

```
eq X:S ==> X = true .
```

is declared. (In fact, a single equation declared in `RWL` works for any `S`.) This axiom states reflexivity. Secondly, for each unconditional tranision

```
trans t => t' .
```

the system declares

```
eq t ==> t' = true .
```

and for each conditional transition

```
ctrans t => t' if c .
```

the system declares

```
ceq t ==> t' = true if c .
```

These declarations correspond to one-step transitions.

Thirdly, for each operator declaration

```
op f : S1 ... Sn -> S
```

the system declares

```
ceq f(X1:S1, ..., Xn:Sn) ==> f(X1':S1, ..., Xn':Sn) = true
   if X1 ==> X1' and ... and Xn ==> Xn' .
```

which states that "_==>_" is closed under f's application.

As to transitivity, a straightforward declaration would not do: an axiom of the form

```
ceq X:S ==> X':S = true if X ==> X'':S and X'' ==> X' .
```

introduces an extra variable in the condition, which makes it inappropriate for the purpose of evaluation (cf. Section 6.1.2). Considering that the main usage of the predicate "==>" is to prove theorems with evaluation mechanism, this fact renders such an axiom useless. In its place the system add the following axiom.

```
ceq X:S ==> X':S = true if X =(*)=> X' .
```

Attached to the operator "_=(*)=>_" is a systematic search procedure for determining whether the first argument transits to the second in an arbitraty number of steps. More on this and related operators in Section 10.2.3.

4.4.3 Behavioural Equivalence Predicate

The predicate for behavioural equivalence, written "=*=", is a binary operator defined on each hidden sort. As with "==" and "==>" in previous sections, an explicit declaration would be of the form

```
 pred _=*=_ : S S
```

for a given sort S.

Remember that behavioural operators are declared with bop, and each such operator has exactly one hidden sort in its arity (Section 3.2.2). On the assumption that "a =*= a'" implies "f1(a) == f1(a')" for any "attribute" f and for any a, if "=*=" turns out to be congruent wrt all the "method"s, the system declares an axiom

```
eq A =*= A' = f1(A) == f1(A') and ... and fn(A) == fn(A') .
```

where f1, ..., fn are the attributes.

Recalling an example of bank accounts

```
*[ Account ]*
bop deposit : Account Nat -> Account
bop withdraw : Account Nat -> Account
bop balance : Account -> Nat
```

the above assumption is

```
ceq balance(A:Account) = balance(A':Account) if A =*= A' .
```

and "=*=" is congruent if two equations

```
eq deposit(a,n) =*= deposit(a',n) = true .
eq withdraw(a,n) =*= withdraw(a',n) = true .
```

hold, on the hypothesis that

```
eq a =*= a' = true .
```

for arbitrary a, a', and n.

The system tries to prove these two equations by evaluation (Chapter 6). If "_=*=_" does turn out to be a congruence, the system adds the axiom

```
ceq A:Account =*= A':Account if balance(A) == balance(A') .
```

Intuitively, this axiom states that two terms of Account are indistinguishable if they are indistuiguishable by a single application of balance.

Note that, if "=*=" is not turned out to be congruent, no axiom defines "=*=". In such a case, "=*=" is not even reflexive.

Another predicate for behavioural equivalence is "=b=," which is to behavioural equations as "==" is to plain ones, and is defined to true iff both hand sides evaluate to a common term. The difference between "==" and "=b=" is in the applicability of behavioural axioms (See 6.2.1 for detail).

4.4.4 Sort Predicate

The system supplies a predicate that checks whether a given term is of a given sort. It is of the form

```
t :is S
```

and evaluates to true if t belongs to S. You can supply explicit axioms for the predicate. An example is

```
vars N N' : Nat
ceq N + N' :is NzNat = true if N :is NzNat .
ceq N + N' :is NzNat = true if N' :is NzNat .
```

Warning This feature is introduced as a surrogate of membership relation. We stress that it is a surrogate, not a real thing. The existence (and implementation) of this predicate is highly tentative, and maybe dropped entirely at any time. You should not use this feature unless absolutely necessary. There are switches such as "mel sort" related to this predicate. They are also highly tentative, and you should not use them unless absolutely necessary.

Chapter 5

Inspecting Modules

It is time we show some examples that combine all of the declarations explained so far. This chapter also illustrates the module inspection command `show` (Section 1.3.3).

5.1 Using Inspection Commands

5.1.1 Printing Whole Modules

We first recall `SIMPLE-NAT`, which defines natural number addition (Section 2.2).

```
module SIMPLE-NAT {
  [ Zero NzNat < Nat ]
  op 0  : -> Zero
  op s : Nat -> NzNat
  op _+_ : Nat Nat -> Nat
  vars N N' : Nat
  eq 0 + N = N .
  eq s(N) + N' = s(N + N') .
}
```

Everything in this module has been already explained, so you should know that this is indeed a standard recursive definition of addition. A subtle point is in the use of subsort relation. Here the set of natural numbers is the union of `Zero`, which contains `0` (and nothing else), and `NzNat`, which contains `s(0)`, `s(s(0))`, and so forth.

Suppose this definition is in the file `simple-nat.mod`. Then you may get the following session.

```
CafeOBJ> input simple-nat

-- processing input : ./simple-nat.mod
-- defining module SIMPLE-NAT....._..* done.
CafeOBJ> show SIMPLE-NAT
module SIMPLE-NAT
{
  imports {
    protecting (BOOL)
  }
  signature {
    [ Nat, Zero NzNat < Nat,
      NzNat, NzNat < Nat,
```

```
      Zero, Zero < Nat ]
    op 0 : -> Zero
    op s : Nat -> NzNat
    op _ + _ : Nat Nat -> Nat { strat: (1 0 2) }
  }
  axioms {
    var N : Nat
    var N' : Nat
    eq 0 + N = N .
    eq s(N) + N' = s(N + N') .
  }
}

CafeOBJ>
```

When invoked with a module name, show command prints the entire contents of the module. You may notice that

- Block constructs imports, signature, and axioms (Section 2.2) are used, even though you did not use them.
- The module BOOL is imported automatically. As explained earlier, BOOL defines Boolean operators and is imported unless prohibited by a switch (Section 2.5).
- There is a bracketed text after the declaration of "_+_". This is an operator attribute, to be explained later (Section 7).

Warning Display of sort declarations is often knotted. This is due to implementation restriction; an "optimal" display is very hard to define anyway.

5.1.2 Print Part of Modules

To print partial contents, such as sort declarations only, of a module, you may supply further arguments to show command. Continuing the above session, you get

```
CafeOBJ> show sorts SIMPLE-NAT
* visible sorts :
  Zero, Zero < Nat
  NzNat, NzNat < Nat
  Nat, Zero NzNat < Nat

CafeOBJ> show ops SIMPLE-NAT
.................................(0).................................
  * rank: -> Zero
.................................(s).................................
  * rank: Nat -> NzNat
...............................(_ + _)...............................
  * rank: Nat Nat -> Nat { strat: (1 0 2) }
    - axioms:
      eq 0 + N = N
      eq s(N) + N' = s(N + N')

CafeOBJ>
```

The arguments sorts, ops, etc. specify the parts of the module to be shown. To summarise the mostly obvious,

`sorts`	sort and subsort declarations
`ops`	operator declarations
`vars`	variable declarations
`axioms`	axiom declarations
`params`	parameter declarations (to be introduced later)
`subs`	import declarations (for 'sub'modules)
`sign`	`sorts` and `ops` combined (i.e. the signature)

It is possible to list declarations in imported modules also. For an example for example's sake,

```
CafeOBJ> module SHADOW-NAT {
  protecting (SIMPLE-NAT)
}

-- defining module SHADOW-NAT._* done.
CafeOBJ> show sorts SHADOW-NAT

CafeOBJ> show all sorts SHADOW-NAT
* visible sorts :
  Zero, Zero < Nat
  NzNat, NzNat < Nat
  Nat, Zero NzNat < Nat

CafeOBJ>
```

`SHADOW-NAT` simply imported declarations of `SIMPLE-NAT` and added nothing. The plain `sorts` argument prints nothing, while `all sorts` prints sort declarations in the imported `SIMPLE-NAT`.

> **Warning** `all` does not print anything in the implicitly imported modules. Even if you import `BOOL` explicitly, the system still refuses to print the declarations in them.

In case of axioms, instead of crying `all` to `show` command, you may change a switch called "`all axioms`", as

```
CafeOBJ> set all axioms on
```

If you are inspecting a particular module, it is convient to set it as current (Section 8.1.1), so that the module name may be omitted.

```
CafeOBJ> select SIMPLE-NAT

SIMPLE-NAT> show sorts
* visible sorts :
  Zero, Zero < Nat
  NzNat, NzNat < Nat
  Nat, Zero NzNat < Nat

SIMPLE-NAT> show ops
..................................(0)..................................
  * rank: -> Zero
..................................(s)..................................
  * rank: Nat -> NzNat
..............................(_ + _)..............................
  * rank: Nat Nat -> Nat
    - attributes:  { strat: (1 0 2) }
    - axioms:
      eq 0 + N = N
```

```
      eq s(N) + N' = s(N + N')

SIMPLE-NAT>
```

Note that the prompt has changed, to indicate what module is current.

It is possible to focus on a single sort, operator or imported module. For example,

```
SIMPLE-NAT> show sort Nat
Sort Nat declared in the module SIMPLE-NAT
  - subsort relation :

     ?Nat
       |
      Nat
    /     \
  Zero  NzNat



SIMPLE-NAT> show op _+_
...............................(_ + _)..............................
  * rank: Nat Nat -> Nat
    - attributes { strat: (1 0 2) }
    - axioms:
      eq 0 + N = N
      eq s(N) + N' = s(N + N')

SIMPLE-NAT>
```

Here the subsort relation is shown graphically. ?Nat is an error sort, and you should ignore it for now (cf. Section 6.6).

A switch controls whether the sort of each variable is displayed. show Continuing the session, you get

```
SIMPLE-NAT> set show var sorts on

SIMPLE-NAT> show op _+_
...............................(_ + _)..............................
  * rank: Nat Nat -> Nat
    - attributes { strat: (1 0 2) }
    - axioms:
      eq 0 + N:Nat = N:Nat
      eq s(N:Nat) + N':Nat = s(N:Nat + N':Nat)

SIMPLE-NAT>
```

Warning This switch does not have a total control. In cases when sorts are not apparent — when an axiom is from an imported module, or when variables are redeclared on the fly, and so on — the system displays them willynilly.

5.1.3 Deep Inspection

If you want more detailed explanations of the module contents, use describe, in place of show. This command is useful when you are evaluating terms (Chapter 6). Since we have not yet explained term evaluation mechanisms, you should not expect to understand the following output. It is enough to know the flavour of difference between show and describe.

```
SIMPLE-NAT> describe op _+_
======================================================================
name                 : (_ + _)
module               : SIMPLE-NAT
number of arguments : 2
default attributes   :
 rewrite strategy    : not specified
 syntax              :
   precedence        : not specified
   computed prec.    : 41
   form              : (arg:41) "+" (arg:41)
 theory              :
----------------------------------------------------------------------
rank                 : ?Nat ?Nat -> ?Nat
  module             : SIMPLE-NAT
  theory             :
  rewrite strategy   : (1 2 0)
  precedence         : 41
  lower oprations    :
    Nat Nat -> Nat
  axioms             :
----------------------------------------------------------------------
rank                 : Nat Nat -> Nat
  module             : SIMPLE-NAT
  theory             :
  rewrite strategy   : (1 0 2 0)
  precedence         : 41
  axioms             :
  - equation
    lhs              : 0 + N:Nat
    rhs              : N:Nat
    top operator     : Nat Nat -> Nat
  - equation
    lhs              : s(N:Nat) + N':Nat
    rhs              : s(N:Nat + N':Nat)
    top operator     : Nat Nat -> Nat

SIMPLE-NAT>
```

5.1.4 Inspecting Records

As explained in Section 3.4.1, a record declaration has the same effects as several related declarations of sorts and operators. You may be anxious to know how the system handles record declarations. If you are, put the record `Date` into a module declaration, and inspect it.

```
CafeOBJ> module DATE {
protecting (SIMPLE-NAT)
record Date {
  year  : Nat
  month : Nat
  day   : Nat
} }

-- defining module DATE.._* done.
CafeOBJ> select DATE
```

```
DATE> show sorts
* visible sorts :
  Date, Date < RecordInstance
  RecordDate, RecordDate < RecordId
  Slotyear, Slotyear < AttrId
  Slotmonth, Slotmonth < AttrId
  Slotday, Slotday < AttrId

DATE>
```

SIMPLE-NAT (Section 2.2) should be familiar. The record Date was indeed declared as a sort, as we stated. But it seems that a lot more sorts are involved. They are used to define the operators. To see what operators were declared, you may inspect further. But hold your breath:

```
DATE> show ops
................................(_ { })................................
  * rank: RecordDate -> Date
    - attributes:  { constr }
..............................(_ { _ })...............................
  * rank: RecordDate Attributes -> Date
    - attributes:  { constr }
................................(Date).................................
  * rank: -> RecordDate
    - attributes:  { constr }
..............................(makeDate)...............................
  * rank: Attributes -> Date
    - axioms:
      eq makeDate(ATTRS-6:Attributes) = #!! (make-record-instance 'date
                                                 attrs-6)
..............................(makeDate)...............................
  * rank: -> Date
    - axioms:
      eq makeDate = #!! (make-record-instance 'date)
................................(year).................................
  * rank: -> Slotyear
    - attributes:  { constr }
................................(year).................................
  * rank: Date -> Nat
    - axioms:
      eq year(RID-5:RecordDate { (VAR-SLOTYEAR-2:Slotyear = VAL-11:Nat ,
          ATTRS-6:Attributes) }) = VAL-11:Nat
..............................(set-year)...............................
  * rank: Date Nat -> Date
    - axioms:
      eq set-year(RID-5:RecordDate { (VAR-SLOTYEAR-2:Slotyear = VAL-11:Nat ,
          ATTRS-6:Attributes) },NEW-VAL-12) = RID-5:RecordDate
          { (VAR-SLOTYEAR-2:Slotyear =  NEW-VAL-12 , ATTRS-6:Attributes) }
```

The print continues for month, set-month and so on. We do not explain the above in detail, and only say that these declarations are necessary for the flexible syntax you may use, and for realising record inheritances. Remember that a record may be written as

```
Date { year = 1984, month = 1, day = 1 }
```

or

```
Date { month = 1, day = 1, year = 1984 }
```

interchangeably; you may also omit uninteresting slots, as

```
Date { year = 1984 }
```

where `month` and `date` disappear; and `set`, `set-year` etc. will be declared for every record that inherits `Date`.

Finally, look at axioms.

```
DATE> show axioms
- Equations
  eq day(RID-5:RecordDate { (VAR-SLOTDAY-4:Slotday = VAL-7:Nat , ATTRS-6:Attributes) }) =
      VAL-7:Nat
  eq set-day(RID-5:RecordDate { (VAR-SLOTDAY-4:Slotday = VAL-7:Nat , ATTRS-6:Attributes) },
      NEW-VAL-8) = RID-5:RecordDate { (VAR-SLOTDAY-4:Slotday = NEW-VAL-8 ,
      ATTRS-6:Attributes) }
```

Again we omitted the rest of the (long) output. The above two equations define `day` and `set-day`. Their simplified forms are

```
eq day (Date { day = D, month = M, year = Y }) = D .
eq set-day (Date { day = D, month = M, year = Y }, D') =
      Date { day = D', month = M, year = Y } .
```

which look exactly like the access functions to `day`.

5.2 Some Typical Modules

5.2.1 Unique Models

When you write a specification of natural numbers (Section 5.1.1), your probable intention is to specify "the" natural numbers. For example, you do not want a singleton $\{0\}$ to constitute a model of your specification (the elements are too few), nor a colossal sum $\mathcal{N} + \mathcal{N} + \ldots + \mathcal{N}$ (too many). You want exactly the set $\{0, 1, 2, \ldots\}$ to be the model of your specification.

In the terminology of CafeOBJ, such a module should be interpreted to have tight denotation: it has exactly one model — the usual term model. A module introduced with "`module!`" is always so interpreted. So perhaps it is more to your taste if we have written

```
module! SIMPLE-NAT {
...
}
```

As another example of tight denotation, consider indeterminate choice (cf. Section 4.3) of natural numbers[1].

```
module! CHOICE-NAT {
  extending (SIMPLE-NAT)
  op _|_ : Nat Nat -> Nat
  vars N N' : Nat
  trans N | N' => N .
  trans N | N' => N' .
}
```

[1]The current implementation imports a builtin sort of the same name, so you need a lot of qualifications for this example to work correctly.

In the model you have in mind,

`0 | s(0)` may transit to `0`

`0 | s(0)` may transit to `s(0)`

but

`0 | s(0)` should not transit to `s(s(0))`

`module!` ensures that the last transition is not in the model. To cope with transition relations, the model must have a 2-categorical structure[2]; otherwise, everything is a straightforward extension to usual algebraic semantics.

5.2.2 Inspecting Transition Axioms

When you declare a transition, a couple of axioms are automatically generated (Section 4.4.2). Let us use `CHOICE-NAT` for illustration.

```
CafeOBJ> module! CHOICE-NAT {
    extending (SIMPLE-NAT)
    op _|_ : Nat Nat -> Nat
    vars N N' : Nat
    trans N | N' => N .
    trans N | N' => N' .
  }

-- defining module! CHOICE-NAT..._._.* done.
CafeOBJ> select CHOICE-NAT

CHOICE-NAT> show axioms
- Equations
  ceq cv1:Nat | cv2:Nat ==> cv3:Nat | cv4:Nat = true if cv1:Nat ==> cv3:Nat and
      cv2:Nat ==> cv4:Nat
  ceq s(cv1:Nat) ==> s(cv2:Nat) = true if cv1:Nat ==> cv2:Nat
  ceq cv1:Nat + cv2:Nat ==> cv3:Nat + cv4:Nat = true if cv1:Nat ==> cv3:Nat and
      cv2:Nat ==> cv4:Nat
  eq N | N' ==> N' = true
  eq N | N' ==> N = true
- Rules
  trans N | N' => N
  trans N | N' => N'

CHOICE-NAT> show subs
extending(SIMPLE-NAT)
protecting(RWL)

CHOICE-NAT>
```

As shown above, congruence axioms are added, and a module called `RWL` is imported automatically. `RWL` is the module that declares "`_==>_`".

Warning By the way, where is the reflexivity axiom? Well, it is hidden in the module `RWL`. The current implementation provides a "universal" reflexivity axiom, and each module does not acquire a new axiom. But you should think as if it did.

[2] In fact, since CafeOBJ does not distinguish transitions between the same two elements, a more restricted structure — essentially, partial orders and monotonic functions — serves as well.

5.2.3 Non-isomorphic Models

There are cases when your favourite models are not unique. Examples abound in defining algebraic structures. Monoids may be defined as

```
module* MONOID {
  [ M ]
  op e : -> M
  op _*_ : M M -> M
  vars X Y Z : M
  eq (X * Y) * Z = X * (Y * Z) .
  eq e * X = X .
  eq X * e = X .
}
```

When you write down the above module, what is intended is not (only) a trivial monoid, the singleton $\{e\}$. But that is exactly the tight denotation of this module, which you would get if it were declared with "`module!`". In this case, a module should accommodate any model so long as it has an associative binary operator with an — i.e., the — identity element. A module introduced with "`module*`" is always interpreted as such.

Another example uses behavioural operators.

```
module* COUNTER {
  protecting (SIMPLE-NAT)
  *[ Counter ]*
  bop read : Counter -> Nat
  bop add : Nat Counter -> Counter
  eq read(add(N:Nat, C:Counter)) = N + read(C) .
}
```

Your intention is to define a counter that is incremented by the operator `add`. Tight denotation would lead to the model where, e.g., for a fixed counter `c`,

```
add(s(0), add(s(s(0)), c))
add(s(s(0)), add(s(0), c))
```

are different elements. Intuitively, this model says that adding 1, then 2, is different from adding 2, then 1. When your sole concern is the return values of `read`, this model works fine (adding 2, then 1 increments the counter by 3; so does adding 1, then 2). But there is no reason to preclude a model where the above two terms denote the same element (the order of addition is immaterial). Hence this module was introduced with "`module*`", as above.

5.2.4 Inspecting Hidden Sorts

Two terms of a hidden sort is equal, or *behaviourally equivalent*, , iff they behave the same under any *behavioural context* (cf. [7]), or *observation*. An observation is a term of a visible sort, which contains exactly one (occurrence of one) variable of the hidden sort. (So an observation is a special derived operator — see Section 8.2.2.) In the example of `COUNTER`, the observations for `Counter` is of the form

```
read(X), read(add(m,X)), read(add(m,add(m',X))), ...
```

where `X` is a variable of sort `Counter`, and `m`, `m'` are arbitrary terms of sort `Nat`. For this special case, therefore, the above definition means that two terms `t`, `t'` of sort `Counter` are behaviourally equal iff the equations

```
eq read(t) = read(t') .
eq read(add(m,t)) = read(add(m,t')) .
eq read(add(m,add(m',t))) = read(add(m,add(m',t'))) .
...
```

hold.

Unlike equality over visible sorts, equality over hidden sorts are unlikely to be determined by straightforward reduction; two terms may well turn out to be behaviourally equivalent, even if they have different irreducible forms. It is still possible, however, to use reduction to *help* prove behavioural equivalence.

As explained in Section 4.4.3, the system associates a binary predicate "_=*=_" to each hidden sort, and declare an axiom. You may inspect the detail, as the following example shows.

```
CafeOBJ> module* COUNTER {
    protecting (SIMPLE-NAT)
    *[ Counter ]*
    bop read : Counter -> Nat
    bop add : Nat Counter -> Counter
    eq read(add(N:Nat, C:Counter)) = N + read(C) .
  }

-- defining module* COUNTER...._.*
* system already proved =*= is a congruence of COUNTER done.
CafeOBJ> select COUNTER

COUNTER> show axioms
- Equations
  eq read(add(N:Nat,C:Counter)) = N:Nat + read(C:Counter)
  ceq hs1:Counter =*= hs2:Counter = true if read(hs1:Counter) ==
      read(hs2:Counter)

COUNTER>
```

In processing COUNTER the system added an equation after checking congruence.

Chapter 6

Evaluating Terms

CafeOBJ has an operational semantics based on term rewriting system (TRS)s. This chapter introduces the very basics of TRSs and explains when a CafeOBJ code constitutes a TRS.

6.1 Term Rewriting System

6.1.1 What is a TRS

This section is an intuitive introduction to the computation model of TRS's. First, a preliminary.

> *Substitution.* A substitution is a set S of pairs (v, t) of a variable v and a (ground) term t, such that no two pairs contain the same variable. If (v, t) is in S, t is said to be the *value* of v under A.
>
> *Instantiation.* Given a substitution S and a term t, replacing each (occurrence of each) variable in t with its value under S is called the instantiation of t by S.
>
> *Matching.* Given a *ground* term t and a term p, if there is a substitution S such that the instantiation of p by S equals t, t is said to be matchable to p. To compute such a substitution is called matching.

Given a set R of rewrite rules, computation with R as TRS proceeds as follows: given a *ground* term t,

1. (pattern matching) Find a rewrite rule in R such that a subterm s of t is matchable to its lefthand side.
2. (rewrite) If such a rewrite rule is found, replace s in t with the instance of its righthand by the matching substitution, and go back to 1. with the resulting new term.
3. If no such rewrite rule is found, terminate.

This procedure is called *evaluation*, or *reduction*, of t. If the procedure terminates, the resultant term is said to be *irreducible*, or *normal.* Note well that this procedure is for ground terms only.

If a rewrite rule is conditional, in addition to matchability, you also have to check the condition. To check the condition, its instance by the matching substitution is evaluated. The rule is applicable only when the condition evaluates to (a term denoting) true. A TRS with conditional rewrite rules is called conditional TRS.

Another refinement of the above procedure is to match terms not only syntactically, but modulo an equational theory E. Such a mathching procedure is called E-matching. A typical equational theory is associativity. Modulo associativity on an operator "`_*_`", for example, "`a * (b * c)`" matches "`(X * b) * Y`", where `X`, `Y` are variables, since the former term is equal to "`(a * b) * c`". CafeOBJ does support this kind of matching (cf. Section 7.1).

There may be cases where more than one rules match with more than one subterms — so the above procedure is, in fact, hardly a procedure at all. Which one to choose, and when, is a significant issue. CafeOBJ allows you to control the selection, operator by operator. See Section 7.4.

6.1.2 How a Module Defines a TRS

The axioms of CafeOBJ are equations and transitions, which define equivalence and transition relations. In operational semantics, these axioms are regarded, uniformly, as directed rewrite rules, as follows. If we denote a rewrite rule by `t -> t'`, for relevant terms `t`, `t'`, and `b`,

> `eq t = t' .` is a rewrite rule `t -> t'`.
>
> `ceq t = t' if b .` is a rewrite rule `t -> t'` with the condition `b`.
>
> `trans t => t' .` is a rewrite rule `t -> t'`.
>
> `ctrans t => t' if b .` is a rewrite rule `t -> t'` with the condition `b`.

Behavioural axioms are translated similarly. Then the TRS defined by a module is the set of axioms regarded as rewrite rules as above. Note that the axioms of the imported modules are included in this definition.

For example, `SIMPLE-NAT` (Section 2.2) defines a TRS that contains two rewrite rules

```
0 + N -> N
s(N) + N' -> s(N + N')
```

where `N`, `N'` are variables of sort `Nat`. Since `BOOL` is implicitly imported, its axioms (as rewrite rules) are also in this TRS. `show` command may be used to list all the rules (see examples in Section 6.2.2).

Two kinds of axioms are excluded from TRS's, solely by syntactic criteria. For an axiom to be a rewrite rule,

(1) Every variable that occurs in the righthand side or in the condition must occur in the lefthand side also. Otherwise the terms under computation may become non-ground.

(2) The lefthand side must not be a single variable. Otherwise every term (of a relevant sort) is matchable, so the computation never stops.

For order-sorted TRS, there might be other constraints, such as sort-decreasing property. But CafeOBJ does not impose such a constraint.

There are two important properties of TRSs. If computation always terminates, the TRS is said to be *terminating*. If, whenever a term is rewritten to two different terms, they can be further rewritten to a same term, the TRS is said to be *confluent*. If terminating and confluent, it is *canonical*.

In general, the declarative meaning of an axiom set differs from the operational one (i.e. TRS). But there *is* a coincidence if we restrict ourselves to tight denotation and equations. If the set, regarded as TRS, is canonical, i.e. terminating and confluent,

> Two (ground) terms are equal iff they are reducible to a common term.

For this reason, it is important to distinguish a subset of a TRS derived only from equations (cf. `reduce` command in Section 6.2.1). It also follows that for loose denotation, reduction per se does not suggest anything of much import. (For theorem proving, it *is* useful — see Section 10.2).

6.2 Do the Evaluation

6.2.1 Evaluation Commands

There are three commands for evaluating terms.

Syntax 49: **reduce** *command*   **reduce** [**in** *module_expression* ":"] *term* "."

Syntax 50: **breduce** *command*   **breduce** [**in** *module_expression* ":"] *term* "."

Syntax 51: **execute** *command*   **execute** [**in** *module_expression* ":"] *term* "."

As is ever the case, the period after a blank is a must. `reduce` can be abbreviated to `red`, `breduce` to `bred`, and `execute` to `exec`. Module expressions are introduced in Section 8.4, and we only consider module names here. These commands evaluate the term with a TRS defined by the given module; or by the current module (Section 8.1.1), if the module expression is omitted.

These three commands differ on (1) which axioms constitute the TRS and (2) applicability of some rewrite rules.

(1)
- For `reduce` or `breduce`, only — possibly conditional, possibly behavioural — equations are taken to constitute the TRS. Transitions are excluded. This ensures that under certain conditions (Section 6.1.2) `reduce` gives a decision procedure for equational theory.
- For `execute`, all the axioms constitute the TRS.

(2)
- For `reduce` or `execute`, a rewrite rule that come from a behavioural axiom is applicable only when the redex in enclosed in an observation. This ensures the soundness of evaluation.
- For `breduce`, there is no such restriction.

The point (1) comes from the fact that `reduce` (`breduce`) is to get a term equal (behaviourally equivalent) to the input, while `execute` is to get a term that the input may eventually transit to. The point (2) comes from the fact that `breduce` is to get a term behaviourally equivalent to the input, while `reduce` is to get a term equal to the input. These points are illustated in Sections 6.2.3 and 6.2.4.

During evaluation, it is sometimes desirable to see the rewrite sequences, not just the results. For example, to enhance efficiency, you may want to know where redundant computations of subterms occur. For this purpose, CafeOBJ provides trace switches, which can be set by

```
CafeOBJ> set trace whole on
```

by which the resultant term of each rewrite step is printed, or

```
CafeOBJ> set trace on
```

by which the rule, substitution, and replacement are printed. The effects of these switches shall be illustrated by examples below.

6.2.2 Replacing Equals by Equals

Recall `SIMPLE-NAT` (Section 2.2) again, make it current, and do some evaluations.

```
SIMPLE-NAT> reduce 0 .

-- reduce in SIMPLE-NAT : 0
0 : Zero
(0.000 sec for parse, 0 rewrites(0.000 sec), 0 matches)
SIMPLE-NAT> reduce 0 + s(0) .

-- reduce in SIMPLE-NAT : 0 + s(0)
s(0) : NzNat
(0.000 sec for parse, 1 rewrites(0.000 sec), 1 matches)

SIMPLE-NAT> reduce s(0) + 0 .

-- reduce in SIMPLE-NAT : s(0) + 0
s(0) : NzNat
(0.010 sec for parse, 2 rewrites(0.000 sec), 3 matches)
SIMPLE-NAT>
```

The output from the system has three parts, showing the system's understanding of what to do (the line starting with "`--`"), the result of evaluation (the second line), and what looks like statistics (in parentheses). The result is of the form

```
term : sort
```

where `sort` is the least sort (Section 3.3.1) of `term`.

You may print the rules of `SIMPLE-NAT` as follows.

```
SIMPLE-NAT> show rules
 -- rewrite rules in module : SIMPLE-NAT
  1 : eq 0 + N = N
  2 : eq s(N) + N' = s(N + N')

SIMPLE-NAT>
```

Note that all the axioms of the implicitly imported `BOOL` are also in the TRS, but were not printed above. It is possible to print all the rules, including those declared in the imported modules, by an extra argument to `show`.

```
SIMPLE-NAT> show all rules
 -- rewrite rules in module : SIMPLE-NAT
  1 : eq 0 + N = N
  2 : eq s(N) + N' = s(N + N')
  3 : eq not true = false
  4 : eq not false = true
  5 : eq false and A:Bool = false
  6 : eq true or A:Bool = true
  7 : eq true and-also A:Bool = A:Bool
  8 : eq A:Bool and-also true = A:Bool
  9 : eq false and-also A:Bool = false
 10 : eq A:Bool and-also false = false
 11 : eq true or-else A:Bool = true
 12 : eq A:Bool or-else true = true
 13 : eq false or-else A:Bool = A:Bool
 14 : eq A:Bool or-else false = A:Bool
 15 : eq A:Bool implies B:Bool = not A:Bool or B:Bool
 16 : eq true xor true = false
```

```
  17 : eq CXU : Id:SortId = #!! (coerce-to-bool
                                   (test-term-sort-membership cxu id))
  18 : eq if true then CXU else CYU fi = CXU
  19 : eq if false then CXU else CYU fi = CYU
  20 : eq CXU == CYU = #!! (coerce-to-bool (term-equational-equal cxu cyu))
  21 : eq HXU =b= HYU = #!! (coerce-to-bool (term-equational-equal hxu hyu))
  22 : eq CXU =/= CYU = #!! (coerce-to-bool
                               (not (term-equational-equal cxu cyu)))

SIMPLE-NAT>
```

Warning The righthand sides of the rules `17` and so on contain a strange symbol `#!!`, and the terms that follow suspiciously resemble Lisp's S-expressions. These are examples of system-dependent definitions of operators. If you are not confident enough about implementation issues, do not touch these definitions.

Since the axioms of `SIMPLE-NAT` are all equations, the results of `reduce` and `execute` should be the same. Indeed, you have

```
SIMPLE-NAT> execute 0 + s(0) .

-- execute in SIMPLE-NAT : 0 + s(0)
s(0) : NzNat
(0.010 sec for parse, 1 rewrites(0.000 sec), 1 matches)
SIMPLE-NAT> execute s(0) + 0 .

-- execute in SIMPLE-NAT : s(0) + 0
s(0) : NzNat
(0.000 sec for parse, 2 rewrites(0.000 sec), 3 matches)
SIMPLE-NAT>
```

The statistics may be suppressed by a switch.

```
SIMPLE-NAT> set stats off

SIMPLE-NAT> reduce s(0) + 0 .

-- reduce in SIMPLE-NAT : s(0) + 0
s(0) : NzNat
SIMPLE-NAT>
```

Warning `stats` is not an abbreviation.

6.2.3 Equations and Transitions

To see the difference between `execute` and `reduce`, we switch to `CHOICE-NAT` (Section 5.2.1).

```
SIMPLE-NAT> select CHOICE-NAT

CHOICE-NAT> set stats on

CHOICE-NAT> reduce s(0) | (s(s(0)) + s(s(s(0)))) .

-- reduce in CHOICE-NAT : s(0) | (s(s(0)) + s(s(s(0))))
s(0) | s(s(s(s(s(0))))) : Nat
(0.010 sec for parse, 3 rewrites(0.000 sec), 5 matches)
```

```
CHOICE-NAT> execute s(0) | (s(s(0)) + s(s(s(0)))) .

-- execute in CHOICE-NAT : s(0) | (s(s(0)) + s(s(s(0))))
s(0) : NzNat
(0.000 sec for parse, 1 rewrites(0.000 sec), 1 matches)

CHOICE-NAT> show all rules
 -- rewrite rules in module : CHOICE-NAT
  1 : ceq cv1:Nat | cv2:Nat ==> cv3:Nat | cv4:Nat = true if cv1:Nat ==>
    cv3:Nat and cv2:Nat ==> cv4:Nat
  2 : ceq s(cv1:Nat) ==> s(cv2:Nat) = true if cv1:Nat ==> cv2:Nat
  3 : ceq cv1:Nat + cv2:Nat ==> cv3:Nat + cv4:Nat = true if cv1:Nat ==>
    cv3:Nat and cv2:Nat ==> cv4:Nat
  4 : eq N | N' ==> N' = true
  5 : eq N | N' ==> N = true
  6 : trans N | N' => N
  7 : trans N | N' => N'
  8 : eq 0 + N = N
  9 : eq s(N) + N' = s(N + N')
 10 : ...
```

We omitted the rest of the listing, which contains axioms in `BOOL`. `reduce` command ignores transition declarations `6` and `7`, so the reduction stopped after rewriting the second argument of `_|_`. `execute` applied the rule `6` immediately, and just one rewrite (see the statistics) led to the result.

Warning Technically, the TRS faithful to the underlying model should be a TRS, derived entirely from transitions, over equivalence classes of terms, determined by equations. The current system does not distinguish transitions and equations during evaluation, and acts faithfully only when certain conditions are satisfied. For detail, see [12].

6.2.4 Using Behavioural Equations

As a matter of course, behavioural equations can be used to prove behavioural equivalence. With judicious use, they can also be used to prove equality. Consider the following example.

```
module* NAT-STREAM {
  protecting (SIMPLE-NAT)
  *[ Stream ]*
  bop __ : Nat Stream -> Stream
  bop hd : Stream -> Nat
  bop tl : Stream -> Stream
  op zeros : -> Stream
  var N : Nat
  var S : Stream
  eq hd(N S) = N .
  beq tl(N S) = S .
  eq hd(zeros) = 0 .
  beq tl(zeros) = zeros .
}
```

A behavioural equation asserts that `tl(n s)` is behaviourally equivalent to `s` for any `n` and `s`, but does not assert that they are equal. Therefore, on the one hand, it is incorrect to reason that `tl(s(0) zeros)` equals `zeros` from this axiom. On the other hand, it is correct to reason that `hd(tl(s(0) zeros))` is equal to `hd(zeros)` and hence to `0`, since `hd`, being an observation on `Stream`, gives the same answer if applied to behaviourally equivalent terms.

In processing **reduce**, the system tries to apply behavioural equations (as rewrite rules) only when the redex is under an observation, as shown below.

```
CafeOBJ> select NAT-STREAM

NAT-STREAM> reduce hd(tl(s(0) zeros)) .

-- reduce in NAT-STREAM : hd(tl(s(0) zeros))
0 : Zero
(0.010 sec for parse, 2 rewrites(0.000 sec), 3 matches)
NAT-STREAM> reduce tl(s(0) zeros) .

-- reduce in NAT-STREAM: tl(s(0) zeros)
tl(s(0) zeros) : Stream
(0.000 sec for parse, 0 rewrites(0.000 sec), 0 matches)
NAT-STREAM>
```

Compare the above result to the case when **breduce** is invoked instead. Since this command is to get an behaviourally equivalent term, the system applies behavioural equations indiscriminately, as

```
NAT-STREAM> breduce tl(s(0) zeros) .

-- behavioural reduce in NAT-STREAM : tl(s(0) zeros)
zeros : Stream
(0.000 sec for parse, 1 rewrites(0.000 sec), 1 matches)
NAT-STREAM>
```

6.2.5 Evaluation Traces

We continue the session of the previous section. The statistics contained the number of rewrites and match attempts. Reduction of "`0 + s(0)`" and "`s(0) + 0`" in Section 6.2.2 amounts to computing $0 + 1$ and $1 + 0$ respectively, and the statistics showed that the efforts of computation were greater with $1 + 0$. You probably know why. To confirm your reasoning, try trace switches.

```
CHOICE-NAT> select SIMPLE-NAT

SIMPLE-NAT> set trace whole on

SIMPLE-NAT> reduce 0 + s(0) .

-- reduce in SIMPLE-NAT : 0 + s(0)
[1]: 0 + s(0)
 ---> s(0)
s(0) : NzNat
(0.000 sec for parse, 1 rewrites(0.000 sec), 1 matches)
SIMPLE-NAT> reduce s(0) + 0 .

-- reduce in SIMPLE-NAT : s(0) + 0
[1]: s(0) + 0
 ---> s(0 + 0)
[2]: s(0 + 0)
 ---> s(0)
s(0) : NzNat
(0.000 sec for parse, 2 rewrites(0.010 sec), 3 matches)
SIMPLE-NAT>
```

The system printed the reduction traces. They show that the first reduction immediately cancelled a right identity zero, while the second reduction, which could not rely on the knowledge that zero is also a left identity, took two rewrites to get the result.

Sometimes more elaborate traces are desirable.

```
SIMPLE-NAT> set trace whole off

SIMPLE-NAT> set trace on

SIMPLE-NAT> reduce s(0) + 0 .

-- reduce in SIMPLE-NAT : s(0) + 0
 1>[1] rule: eq s(N:Nat) + N':Nat = s(N:Nat + N':Nat)
     { N':Nat |-> 0, N:Nat |-> 0 }
 1<[1] s(0) + 0 --> s(0 + 0)
 1>[2] rule: eq 0 + N:Nat = N:Nat
     { N:Nat |-> 0 }
 1<[2] 0 + 0 --> 0
s(0) : NzNat
(0.000 sec for parse, 2 rewrites(0.000 sec), 3 matches)
SIMPLE-NAT>
```

This time, each rewrite step was explained in more detail. The three lines starting with "1>" correspond to a single step. They show the rewrite rule (i.e. equation or transition) applied, the substitution used, and the replacement.

"trace whole" switch traces the entire term, while replacements shown by trace switch focus on the changed subterms. This difference was illustrated by the last step of this example, where "trace whole" printed

```
[2] s(0 + 0) --> s(0)
```

while trace printed

```
[2] 0 + 0 --> 0
```

6.2.6 Examples of Conditionals

To illustrate the rôle of conditions, we first define an operator that computes greatest common divisors by filling more meat in the bony SIMPLE-NAT.

```
module SIMPLE-NAT+ {
  protecting (SIMPLE-NAT)
  op _-_ : Nat Nat -> Nat
  op _<_ : Nat Nat -> Bool
  -- ---------------------
  vars N M : Nat
  eq 0 - M = 0 .
  eq N - 0 = N .
  eq s(N) - s(M) = N - M .
  eq 0 < s(N) = true .
  eq N < 0 = false .
  eq s(N) < s(M) = N < M .
}
```

```
module GCD {
  protecting (SIMPLE-NAT+)
  op gcd : Nat Nat -> Nat
  -- --------------------
  vars N M : Nat
  ceq gcd(N, M) = gcd(M, N) if N < M .
  eq gcd(N, 0) = N .
  ceq gcd(s(N), s(M)) = gcd(s(N) - s(M), s(M))
      if not (N < M) .
}
```

The definitions of the pseudo-subtraction and the less than relation are more or less standard. gcd is defined by conditional equations. Note that BOOL is imported implicitly, so that all the operators in it, like "not_" here, are available.

To see how these definitions work, continue the session in the previous section. Feed the modules to the system, and try as follows.

```
SIMPLE-NAT> set trace off

SIMPLE-NAT> select GCD

GCD> reduce gcd(s(s(s(s(s(s(0)))))), s(s(s(s(0))))) .

-- reduce in GCD : gcd(s(s(s(s(s(s(0)))))),s(s(s(s(0)))))
s(s(0)) : NzNat
(0.000 sec for parse, 44 rewrites(0.010 sec), 113 matches)
GCD>
```

The reduction amounts to computing the greatest common divisor of 6 and 4. The statistics show that the system worked much harder than ever. You may get traces as before. Since the output is largish, we show traces of simpler computations (but this is still large).

```
GCD> set trace whole on

GCD> reduce gcd(s(s(s(s(0)))), s(s(0))) .

-- reduce in GCD : gcd(s(s(s(s(0)))),s(s(0)))
[1(cond)]: s(s(s(s(0)))) < s(s(0))
   --> s(s(s(0))) < s(0)
[2(cond)]: s(s(s(0))) < s(0)
   --> s(s(0)) < 0
[3(cond)]: s(s(0)) < 0
   --> false
[4(cond)]: not s(s(s(0))) < s(0)
   --> not s(s(0)) < 0
[5(cond)]: not s(s(0)) < 0
   --> not false
[6(cond)]: not false
   --> true
[7]: gcd(s(s(s(s(0)))),s(s(0)))
 ---> gcd(s(s(s(s(0)))) - s(s(0)),s(s(0)))
[8]: gcd(s(s(s(s(0)))) - s(s(0)),s(s(0)))
 ---> gcd(s(s(s(0))) - s(0),s(s(0)))
[9]: gcd(s(s(s(0))) - s(0),s(s(0)))
 ---> gcd(s(s(0)) - 0,s(s(0)))
[10]: gcd(s(s(0)) - 0,s(s(0)))
```

```
 ---> gcd(s(s(0)),s(s(0)))
[11(cond)]: s(s(0)) < s(s(0))
   --> s(0) < s(0)
[12(cond)]: s(0) < s(0)
   --> 0 < 0
[13(cond)]: 0 < 0
   --> false
[14(cond)]: not s(0) < s(0)
   --> not 0 < 0
[15(cond)]: not 0 < 0
   --> not false
[16(cond)]: not false
   --> true
[17]: gcd(s(s(0)),s(s(0)))
 ---> gcd(s(s(0)) - s(s(0)),s(s(0)))
[18]: gcd(s(s(0)) - s(s(0)),s(s(0)))
 ---> gcd(s(0) - s(0),s(s(0)))
[19]: gcd(s(0) - s(0),s(s(0)))
 ---> gcd(0 - 0,s(s(0)))
[20]: gcd(0 - 0,s(s(0)))
 ---> gcd(0,s(s(0)))
[21(cond)]: 0 < s(s(0))
   --> true
[22]: gcd(0,s(s(0)))
 ---> gcd(s(s(0)),0)
[23(cond)]: s(s(0)) < 0
   --> false
[24]: gcd(s(s(0)),0)
 ---> s(s(0))
s(s(0)) : NzNat
(0.000 sec for parse, 24 rewrites(0.070 sec), 60 matches)
GCD>
```

The apparent difference from the previous traces is the presence of "`[(#)cond]`" steps. As explained in Section 6.1.1, the system reduces conditions when it checks applicability of conditional rewrite rules. The above trace was the result of following all of those recursive evaluations. For brevity we now resort to a very simple computation, and show only the initial part of the initial trace.

```
GCD> set trace whole off

GCD> set trace on

GCD> reduce gcd(s(s(0)),s(s(0))) .

 -- reduce in GCD : gcd(s(s(0)),s(s(0)))
 1>[1] apply trial #1
 -- rule: ceq gcd(N:Nat,M:Nat) = gcd(M:Nat,N:Nat) if N:Nat < M:Nat
     { M:Nat |-> s(s(0)), N:Nat |-> s(s(0)) }
 2>[1] rule: eq s(N:Nat) < s(M:Nat) = N:Nat < M:Nat
     { M:Nat |-> s(0), N:Nat |-> s(0) }
 2<[1] s(s(0)) < s(s(0)) --> s(0) < s(0)
 2>[2] rule: eq s(N:Nat) < s(M:Nat) = N:Nat < M:Nat
     { M:Nat |-> 0, N:Nat |-> 0 }
 2<[2] s(0) < s(0) --> 0 < 0
 2>[3] rule: eq N:Nat < 0 = false
     { N:Nat |-> 0 }
 2<[3] 0 < 0 --> false
```

```
1>[4] -- rewrite rule exhausted (#1)
1>[4] apply trial #2
-- rule: ceq gcd(s(N:Nat),s(M:Nat)) = gcd(s(N:Nat) - s(M:Nat),s(M:Nat)
      ) if not N:Nat < M:Nat
    { M:Nat |-> s(0), N:Nat |-> s(0) }
2>[4] rule: eq s(N:Nat) < s(M:Nat) = N:Nat < M:Nat
    { M:Nat |-> 0, N:Nat |-> 0 }
2<[4] s(0) < s(0) --> 0 < 0
2>[5] rule: eq N:Nat < 0 = false
    { N:Nat |-> 0 }
2<[5] 0 < 0 --> false
2>[6] rule: eq not false = true
    {}
2<[6] not false --> true
1>[7] match success #2
1<[7] gcd(s(s(0)),s(s(0))) --> gcd(s(s(0)) - s(s(0)),s(s(0)))
```

Note the appearances of "2>". The number indicates the depth of recursion: evaluating conditions may lead to recursively evaluating conditions, and so on and on. Most of the output should be self-explanatory. The above trace shows two conditional rules were checked, first unsuccessfully and second successfully.

6.3 Stepper

By taking traces, you may inspect in detail how reductions proceed, and modify axioms if they are not going as expected. The set of commands introduced in this section is for controlling reductions in more detail, and help you "debug" modules.

6.3.1 Step Mode

You may enter into/leave the step mode by a switch.

```
CafeOBJ> set step on
```

In this mode, evaluation commands apply rewrite rules step by step. We first define multiplication.

```
module MULT {
  protecting (SIMPLE-NAT)
  op _*_ : Nat Nat -> Nat
  vars N M : Nat
  eq 0 * N = 0 .
  eq s(N) * M = M + (N * M) .
}
```

After feeding `SIMPLE-NAT` and `MULT`, you may get the following session.

```
CafeOBJ> select MULT

MULT> set step on

MULT> reduce s(s(0)) * s(s(s(0))) .

-- reduce in MULT : s(s(0)) * s(s(s(0)))
>> stepper term: s(s(0)) * s(s(s(0)))
STEP[1]?
```

The new prompt indicates you are in the step mode. You may list the commands available in this mode by the help command "`:?`".

```
STEP[1]? :?
-- Stepper command help :
  ?                print out this help
  n(ext)           go one step
  g(o) <number>    go <number> step
  c(ontinue)       continue rewriting without stepping
  q(uit)           leave stepper continuing rewrite
  a(bort)          abort rewriting
  r(ule)           print out current rewrite rule
  s(subst)         print out substitution
  l(imit)          print out rewrite limit count
  p(attern)        print out stop pattern
  stop [<term>] .  set(unset) stop pattern
  rwt [<number>] .set(unset) max number of rewrite
-- the followings are subset of CafeOBJ interpreter commands
  show -or-
  describe         print various info., for further help, type 'show ?'
  set              set toplevel switches, for further help: type 'set ?'
  cd <directory>   change current directory
  ls <directory>   list files in directory
  pwd              print current directory
  lisp -or-
  lispq <lisp>     evaluate lisp expression <lisp>
  ! <command>      fork shell <command>. Under Unix only
STEP[1]?
```

Some of them, such as `show`, `set`, and `cd`, should be familiar. They are not particular to the step mode. Particular to the mode are the commands prefixed by "`:`", in addition to `rwt` and `stop`.

The prefix "`:`" may be omitted, and there are longer forms for some step mode commands. Here is a summary of synonyms.

:?, :h	?, h	:help	help
:n	n	:next	next
:g	g	:go	go
:c	c	:continue	continue
:q	q		
:a	a	:abort	abort
:r	r	:rule	rule
:s	s	:subst	subst
:l	l	:limit	limit
:p	p	:pattern	pattern
:rwt	rwt	:rewrite	rewrite
		:stop	stop

Warning The command names are tentative, and are subject to change.

6.3.2 Evaluation Under the Step Mode

Continuing the session in the previous section, we illustrate the effects of each command.

```
STEP[1]? :n
```

```
>> stepper term: s(s(s(0))) + (s(0) * s(s(s(0))))
STEP[2]? :n

>> stepper term: s(s(0)) + (s(0) * s(s(s(0))))
STEP[3]? :n
>> stepper term: s(0) + (s(0) * s(s(s(0))))
STEP[4]?
```

":n" prints a kind of traces, but beware that the printed term is a subterm under replacement. You may print the current form of the entire term as

```
STEP[4]? show term
s(s((s(0) + (s(0) * s(s(s(0))))))) : NzNat
STEP[4]?
```

Or you may get a better (if longer) trace if that is what you want, by setting "`trace whole`" to `on`. Starting from the first step, the output is

```
MULT> set trace whole on

MULT> reduce s(s(0)) * s(s(s(0))) .

-- reduce in MULT : s(s(0)) * s(s(s(0)))
>> stepper term: s(s(0)) * s(s(s(0)))
STEP[1]? :n

[1]: s(s(0)) * s(s(s(0)))
 ---> s(s(s(0))) + (s(0) * s(s(s(0))))
>> stepper term: s(s(s(0))) + (s(0) * s(s(s(0))))
STEP[2]? :n

[2]: s(s(s(0))) + (s(0) * s(s(s(0))))
 ---> s(s(s(0)) + (s(0) * s(s(s(0)))))
>> stepper term: s(s(0)) + (s(0) * s(s(s(0))))
STEP[3]? :n

[3]: s(s(s(0)) + (s(0) * s(s(s(0)))))
 ---> s(s(s(0) + (s(0) * s(s(s(0))))))
>> stepper term: s(0) + (s(0) * s(s(s(0))))
STEP[4]?
```

which shows the stepper is focusing more and more inside. To skip several steps, use `:g`.

```
STEP[4]? set trace whole off

STEP[4]? :g 5

>> stepper term: s(0) + (0 * s(s(s(0))))
STEP[9]? :g 5

s(s(s(s(s(0))))) : NzNat
(0.010 sec for parse, 11 rewrites(0.040 sec), 19 matches)
MULT>
```

`show` command may take arguments `term` *and* `tree`. Then a tree form of the term is printed (See Section 9.2.3 for an example).

The rest of the commands act as follows.

```
MULT> reduce s(s(0)) * s(s(s(0))) .

-- reduce in MULT : s(s(0)) * s(s(s(0)))
>> stepper term: s(s(0)) * s(s(s(0)))
STEP[1]? :a

s(s(0)) * s(s(s(0))) : Nat
(0.000 sec for parse, 1 rewrites(0.000 sec), 2 matches)
MULT>
```

"`:a`" stops the evaluation immediately.

```
MULT> reduce s(s(0)) * s(s(s(0))) .

-- reduce in MULT : s(s(0)) * s(s(s(0)))
>> stepper term: s(s(0)) * s(s(s(0)))
STEP[1]? :n

>> stepper term: s(s(s(0))) + (s(0) * s(s(s(0))))
STEP[2]? :r
(s(N:Nat) + N':Nat) = s((N:Nat + N':Nat))
STEP[2]? :s
{ N':Nat |-> s(0) * s(s(s(0))), N |-> s(s(0)) }
STEP[2]? :c

s(s(s(s(s(0))))) : NzNat
(0.000 sec for parse, 11 rewrites(0.010 sec), 19 matches)
MULT>
```

"`:r`" and "`:s`" show the rule and the substitution to be used in the *next* step (study the above example carefully). "`:c`" continues the evaluation non-stop.

```
MULT> reduce s(s(0)) * s(s(s(0))) .

-- reduce in MULT : s(s(0)) * s(s(s(0)))
>> stepper term: s(s(0)) * s(s(s(0)))
STEP[1]? :q

s(s(s(s(s(0))))) : NzNat
(0.000 sec for parse, 11 rewrites(0.010 sec), 19 matches)
MULT> reduce s(s(0)) * s(0) .

-- reduce in MULT : s(s(0)) * s(0)
s(s(0)) : NzNat
(0.000 sec for parse, 7 rewrites(0.000 sec), 11 matches)
MULT>
```

"`:q`" forces the system to leave the step mode, as the above example shows.

For "`:l`" and "`:p`", see the next section.

6.3.3 Controlled Reduction, by Patterns

It is possible to stop reductions when a certain pattern is found.

Syntax 52: `stop` *command*
  `stop` *term* "`.`"
  `stop` "`.`"

where a term is a term in the current module context (Section 8.1.1), and may contain variables. Do not forget the last blank-period stopper. 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. Alternatively, you may use `set` command, as

```
CafeOBJ> set stop pattern t .
```

where `t` is a term.

Warning This command is fast becoming obsolete, and it is advisable to use the above `set` command instead.

We continue the session with `MULT` (Note that the `step` switch is still `on`).

```
MULT> set stop pattern 0 * N:Nat .

MULT> reduce s(s(0)) * s(s(s(0))) .

-- reduce in MULT : s(s(0)) * s(s(s(0)))
>> stepper term: s(s(0)) * s(s(s(0)))
STEP[1]? :p

[stop pattern]: (0 * N:Nat)
STEP[1]? :c

>> subterm : 0 * s(s(s(0)))
   of term : s(s(s(s(s(s(0))) + (0 * s(s(s(0)))))))
   matches to stop pattern: 0 * N
<< will stop rewriting
>> stop because matches stop pattern.
>> stepper term: s(s(s(0))) + (0 * s(s(s(0))))
STEP[7]? set stop pattern .

STEP[7]? :c

s(s(s(s(s(s(0)))))) : NzNat
(0.010 sec for parse, 11 rewrites(0.030 sec), 19 matches)
MULT>
```

"`:p`" prints the term given by `set` command, and "`:c`" resumes reduction. The system stopped when a matching term appeared, as requested. After the restriction was lifted, the reduction continued uninterrupted.

You may print the current pattern as

```
STEP[1]? show stop
[stop pattern]: (0 * N:Nat)
STEP[1]?
```

instead of :p. This command can be invoked without entering the stepper. In fact, pattern restrictions may be used outside the stepper. For example,

```
MULT> set step off
```

```
MULT> set stop pattern 0 * N:Nat .

MULT> show stop
[stop pattern]: (0 * N:Nat)

MULT> reduce s(s(0)) * s(s(s(0))) .

-- reduce in MULT : s(s(0)) * s(s(s(0)))
>> subterm : 0 * s(s(s(0)))
   of term : s(s(s(s(s(s(0))) + (0 * s(s(s(0)))))))
   matches to stop pattern: 0 * N
<< will stop rewriting
s(s(s(s(s(s(0))) + (0 * s(s(s(0))))))) : NzNat
(0.000 sec for parse, 6 rewrites(0.010 sec), 12 matches)
MULT>
```

The pattern restriction forced the reduction to terminate. As shown above, `show` command with `stop` argument prints the specified pattern. You should use this command, instead of "`:p`" (the latter is recognised only within the stepper).

Warning It is not possible to specify more than one patterns, or more complicated patterns using e.g. regular expressions. If demands are strong, the future versions may enhance the pattern description facility.

6.3.4 Controlled Reduction, by Number of Steps

It is possible to control reductions by limiting rewrite steps. Enter the stepper again.

```
MULT> set step on

MULT> set stop pattern .

MULT> reduce s(s(0)) * s(s(s(0))) .

-- reduce in MULT : s(s(0)) * s(s(s(0)))
>> stepper term: s(s(0)) * s(s(s(0)))
STEP[1]? rwt 10

STEP[1]? :c

>> aborting rewrite due to rewrite count limit (= 10) <<
s(s(s(s(s(s(0 * s(s(s(0))))))))) : NzNat
(0.000 sec for parse, 10 rewrites(0.020 sec), 18 matches)
MULT>
```

`rwt` command in the stepper forced the reduction to abort, after 10 rewrite steps. Note that, unlike "`:g`" command, the system aborted the reduction, not just suspended it.

It is possible to limit rewrite steps outside the stepper.

```
MULT> set step off

MULT> set rwt limit 5

MULT> reduce s(s(0)) * s(s(s(0))) .
```

```
-- reduce in MULT : s(s(0)) * s(s(s(0)))
>> aborting rewrite due to rewrite count limit (= 5) <<
s(s(s(s(0) * s(s(s(0)))))) : NzNat
(0.000 sec for parse, 5 rewrites(0.000 sec), 9 matches)
MULT>
```

You should use `set` command, instead of `rwt` command (the latter is recognised only in the stepper). To lift the restriction, set the limit to `.`

```
MULT> set rwt limit .

MULT> show limit
[rewrite limit]: not specified.

MULT> reduce s(s(0)) * s(s(s(0))) .

-- reduce in MULT : s(s(0)) * s(s(s(0)))
s(s(s(s(s(s(0)))))) : NzNat
(0.000 sec for parse, 11 rewrites(0.000 sec), 19 matches)
MULT>
```

As shown above, `show` command with `limit` argument prints the specified limit.

If both the pattern and limit restrictions are in force, the system aborts reductions at the earliest opportunity.

6.4 Faster Evaluation

Such commands as `reduce` treat terms and rewrite rules as Lisp objects, and the rewrite procedure manipulates these objects directly. If the rule set is huge, and/or the term is or will grow large, the system may become uncomfortably slow. A way to overcome the discomfort is to invest in a faster machine with a larger memory, of course, but there is another, cheaper way. For faster evaluation, you may encode rewrite rules into an abstract machine, and reduce terms by executing that machine, with the following command.

Syntax 53: `tram` *command for evaluation*
`tram` { `reduce` | `execute` } [`in` *module_expression* "`:`"] *term* " " "`.`"

`reduce`, `execute` may be abbreviated to `red`, `exec` respectively. The command acts like `reduce` or `execute` and reduces the given term. If the module expression is omitted, the reduction occurs in the current module.

Warning The compiled code does not distinguish behavioural axioms as such, and `reduce` option acts like `breduce` command.

Upon accepting the command, the system

1. Compiles the rewrite rules of the module into an abstract machine, if they have not been compiled already; Then it
2. Runs the abstract machine on the given term.

The output you get is more or less the same as when `reduce` or `execute` commands are used. For example, with `MULT` (Section 6.3.1), you have

```
MULT> tram reduce s(s(s(0))) * (s(s(s(s(s(0))))) * s(s(s(s(s(0)))))) .
-- reduce in MULT : s(s(s(0))) * (s(s(s(s(s(0))))) * s(s(s(s(0)
```

```
    )))))
s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(
    s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(
    s(s(s(s(s(s(s(s(s(s(s(0))))))))))))))))))))))))))))))))))))))
    ))))))))))))))))))))))))))))))))))))) : NzNat
(rtime 3 utime 3 stime 0 rewrite 118 gc 0)

MULT>
```

For comparison, the corresponding ordinary evaluation works as follows.

```
MULT> reduce s(s(s(0))) * (s(s(s(s(s(0))))) * s(s(s(s(s(0)))))) .

-- reduce in MULT : s(s(s(0))) * (s(s(s(s(s(0))))) * s(s(s(s(s(0)
    )))))
s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(
  s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(
  s(s(s(s(s(s(s(s(s(s(0))))))))))))))))))))))))))))))))))))))))))
  ))))))))))))))))))))))))))))))))) : NzNat
(0.010 sec for parse, 118 rewrites(0.020 sec), 226 matches)
MULT>
```

Compilation is done only once, when `tram` command is first used on the given module.

You may just compile a module as a preparation, by

Syntax 54: `tram` *command for compilation* `tram compile` [`-exec`] [*module_expression*] " " "."

If the module expression is omitted, the current module is assumed. If `-exec` is given, transitions as well as equations are compiled (i.e., for `exec`). For this command, `-e`, `-all`, and `-a` are all synonymous to `-exec`.

Warning `reduce` and `execute` use different sets of rules. Therefore, if you have compiled a module already for `reduce` and try `tram` with `execute` option, the system recompile the module.

Warning Do not forget the last period preceded by a blank.

Compilation into and execution of abstract machines are taken care of by an indepent program, called `tram` or `brute`. So you have to tell the system where it is. The command

```
CafeOBJ> set tram path /home/irene/bin/tram
```

directs the system to the given path for this program. The default is

```
/usr/local/bin/tram
```

`tram` and `brute` have quite different architectures and they both have limitations. For detail, see README in the distribution package.

There is another switch related to `tram`, although you may never use it. The command

```
CafeOBJ> set tram options someoptionsoftram
```

makes the system supply `someoptionsoftram` as options whenever `tram` is invoked. In normal usage, you need not know which options `tram` or `brute` takes.

6.5 Context Variable

To help ease reading and writing terms, a *context variable* facility is supported.

> **Syntax 55:** `let` *command*
> `let` *identifier* "=" *term* "."

The last period preceded by a blank is a must. An identifier is a character string and a term is a term in the current context (Section sec:p2-current-module). This command binds the given term to the identifier in the current context. The example below illustrates the usage, and shows how to print the binding state.

```
MULT> let a1 = s(0) + (s(s(0)) + s(s(s(0)))) .
-- setting let variable "a1" to :
    (s(0) + (s(s(0)) + s(s(s(0))))) : Nat

MULT> let a2 = s(s(s(0))) * (a1 * s(s(s(s(s(0)))))) .
-- setting let variable "a2" to :
    (s(s(s(0))) * ((s(0) + (s(s(0)) + s(s(s(0))))) * s(s(s(s(s(0)))))))
     : Nat

MULT> reduce a1 .

-- reduce in MULT : s(0) + (s(s(0)) + s(s(s(0))))
s(s(s(s(s(s(0)))))) : NzNat
(0.000 sec for parse, 5 rewrites(0.010 sec), 8 matches)
MULT> reduce a2 .


-- reduce in MULT : s(s(s(0))) * ((s(0) + (s(s(0)) + s(s(s(0)))))
     * s(s(s(s(s(0))))))
s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(
  s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(
  s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(0))))))))))))
  )))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))
  ))))))))))))))) : NzNat
(0.000 sec for parse, 145 rewrites(0.010 sec), 277 matches)
MULT> show term a1
(s(0) + (s(s(0)) + s(s(s(0))))) : Nat

MULT>
```

A context variable, once bound, may be used as a subterm in commands `reduce`, `parse`, `let` itself, etc. Note that `let` binds syntactically. `reduce` commands above did not change the values of `a1` or `a2`. `show` command was used to print the binding of `a1`. To list all the bindings, use `show` command again:

```
MULT> show let
[bindings]
a2 = s(s(s(0))) * ((s(0) + (s(s(0)) + s(s(s(0))))) * s(s(s(s(s(0))))))
a1 = s(0) + (s(s(0)) + s(s(s(0))))

MULT>
```

`let` in the above command may be replaced by `binding`.

Context variables are local to each context. Continuing the above session,

```
MULT> select SIMPLE-NAT

SIMPLE-NAT> let a1 = 0 .
-- setting let variable "a1" to :
    0 : Zero

SIMPLE-NAT> show let
[bindings]
a1 = 0

SIMPLE-NAT> select MULT

MULT> show let
[bindings]
a2 = s(s(s(0))) * ((s(0) + (s(s(0)) + s(s(s(0))))) * s(s(s(s(0)))))
a1 = s(0) + (s(s(0)) + s(s(s(0))))

MULT>
```

As shown above, context variables in `MULT` and `SIMPLE-NAT` are kept separately, and the same name may be used for different bindings.

Warning As explained, context variables belong to individual modules. If you attempt to bind a context variable without setting a current module, the system complains — it has no way to know whither the term came.

By the way, you encountered not so small a term when `a2` was reduced. It is very simple for the system to print such terms accurately, they may sometimes cause you nausea or headache. If you expect the terms to be large, and need to see only their broad outline, you can use a switch.

```
CafeOBJ> set print depth 10
```

tells the system to print terms up to the depth 10 (as trees). The default is "unlimited", and the depth of "`.`" is interpreted as such. Continuing the above session, then, you get

```
MULT> set print depth 10

MULT> reduce a2 .

-- reduce in MULT : s(s(s(0))) * ((s(0) + (s(s(0)) + s(s(s(0)))))
     * s(s(s(s(s(0))))))
s(s(s(s(s(s(s(s(s( ... )))))))))) : NzNat
(0.000 sec for parse, 145 rewrites(0.010 sec), 277 matches)
MULT> set print depth .

MULT> reduce a2 .

-- reduce in MULT : s(s(s(0))) * ((s(0) + (s(s(0)) + s(s(s(0)))))
     * s(s(s(s(s(0))))))
s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(
  s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(
  s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(0))))))))))))
  )))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))
  ))))))))))))))) : NzNat
(0.000 sec for parse, 145 rewrites(0.010 sec), 277 matches)
MULT>
```

6.6 Flexible Typing and Error Handling

CafeOBJ has a strong typing discipline. Although useful for static analyses, strong typing is sometimes too restrictive. There are cases where type mismatch could never happen, even though expressions are syntactically illegal. Or, rephrasing in the lexicon of TRS's, apparently ill-formed terms may reduce to well-formed terms. CafeOBJ appraises such phenomena, to offer a "reasonably" strong typing discipline.

Consider a case when you try to apply an operator

```
op f : S -> S'
```

to a term of sort T. S and T differ, so such an attempt should be rejected. Unless, that is, the relation

```
S < T
```

holds. Note well that, if

```
T < S
```

holds, f can definitely apply to terms of T. That is an essence of order-sorted logic. What we are saying here is that, if S is a subsort of T and a term t is of sort T, f(t) may become (by evaluation) a well-formed term, since t may reduce to a term t' of sort S.

One way to accommodate the above case is to use so-called retract operators (c.f. [9]). CafeOBJ has adopted another approach. Let us see how it works.

6.6.1 Factorial of Rationals (!?)

A usual definition of the factorial is as follows.

```
module FACT {
 protecting (RAT)
 op _! : Nat -> NzNat
 eq 0 ! = 1 .
 ceq N:Nat ! = N * (N - 1 !) if N > 0 .
}
```

where the factorial is declared as a postfix operator _!, as is the convention. RAT is a built-in module of rationals. You may point out that rationals are irrelevant to the factorial. They are, and this module is certainly artificial: it is only declared to show that we can compute the factorial of rationals. Well, sometimes.

The sort ordering in RAT defines the graph

$$
\begin{tikzcd}
 & \mathtt{Rat} \arrow[dl] \arrow[d] \\
\mathtt{NzRat} \arrow[d] & \mathtt{Int} \arrow[dl] \arrow[d] \\
\mathtt{NzInt} \arrow[d] & \mathtt{Nat} \arrow[dl] \arrow[d] \\
\mathtt{NzNat} & \mathtt{Zero}
\end{tikzcd}
$$

All these sorts are of familiar kind: `Rat`, `Int`, `Nat` are the sorts of rationals, integers, and natural numbers respectively, and `Nz` means non-zero. `Zero` is a one-point set, containing 0. Unlike our `SIMPLE-NAT`, naturals may be written 1, 43, 107, etc. in this built-in module.

`RAT` defines the division

```
op _/_ : Rat NzRat -> Rat
```

Now

```
(I:Int / J:Int)!
```

is ill-formed, since `I / J` is a rational, while `_!` takes only natural number arguments. But CafeOBJ parses it as well-formed, as follows.

```
CafeOBJ> parse in FACT : (I:Int / J:Int)! .

((I / J) !) : ?Rat
```

Recall that the result of parsing is a pair of a parenthesised term and its least sort. Here the least sort is shown to be a questionable `?Rat`. The naming convention is implementation-dependent and should not bother you, but you should be bothered to know to that the name denotes an *error sort* over `Rat`, i.e. a sort consisting of "error" elements as well as rationals.

For any integers `I` and `J`, then, `(I / J)!` is well-formed, although its sort has a dubious name. It follows that

```
(4 / 2)!
```

is well-formed. Try parsing, then evaluating the term, and you get

```
CafeOBJ> select FACT

FACT> parse (4 / 2)! .

((4 / 2) !) : ?Rat
FACT> reduce (4 / 2)! .
```

```
-- reduce in FACT : 4 / 2 !
2 : NzNat
(0.000 sec for parse, 14 rewrites(0.020 sec), 24 matches)
FACT>
```

The original term is parsed as a term of sort `?Rat`, as before, and the result of evaluation is a term 2 of sort `NzNat`. Indeed, this result is the 2's factorial, i.e. 4/2's factorial. And `NzNat` is not dubious.

On the other hand, if you supply a meaningless term, the result remains dubious.

```
FACT> reduce (4 / 3)! .

-- reduce in FACT : 4 / 3 !
4/3 ! : ?Rat
(0.010 sec for parse, 1 rewrites(0.000 sec), 1 matches)
FACT>
```

To summarise, in CafeOBJ,

(1) If a term is ill-formed but potentially well-formed, it is regarded as a well-formed term of questionable sort.

(2) On evaluation, the potential is investigated. If the result is a term which is unquestionably well-formed, it acquires a full citizenship. Otherwise it remains an outcast.

6.6.2 Stacks

Stacks are ubiquitous topics in our community, especially in discussing exception/error handling. The main reason is that their definition is compact, yet reveals (or conceals) how `top(empty)`, a raving exception, is treated. The module below declares a sort of non-empty stacks, and `top` and `pop` are declared on that sort, not on the sort of all the stacks. In effect, `top` and `pop` are partial operators.

```
module STACK-OF-NAT {
 [ NeStack < Stack ]
 protecting (SIMPLE-NAT)
 op empty : -> Stack
 op push : Nat Stack -> NeStack
 op top_ : NeStack -> Nat
 op pop_ : NeStack -> Stack
 eq top push(X:Nat, S:Stack) = X .
 eq pop push(X:Nat, S:Stack) = S .
}
```

"`top push(s(s(0)), empty)`" is well-formed. It parses and reduces as

```
CafeOBJ> select STACK-OF-NAT

STACK-OF-NAT> parse top push(s(s(0)), empty) .

(top push(s(s(0)),empty)) : Nat
STACK-OF-NAT> reduce top push(s(s(0)), empty) .

-- reduce in STACK-OF-NAT : top push(s(s(0)),empty)
s(s(0)) : NzNat
(0.000 sec for parse, 1 rewrites(0.000 sec), 1 matches)
STACK-OF-NAT>
```

Without introducing error sorts, the term "`top pop push(s(0), push(s(s(0)), empty))` is ill-formed: `top` requires non-empty stacks as arguments, but the coarity of `pop` is `Stack`, which contains the empty stack. CafeOBJ regards this term, like `(4 / 2)!`, as well-formed, but of an error sort `?Nat`. On evaluation, it reduces to a well-formed term of a valid sort.

```
STACK-OF-NAT> parse top pop push(s(0), push(s(s(0)), empty)) .

(top (pop push(s(0),push(s(s(0)),empty)))) : ?Nat
STACK-OF-NAT> reduce top pop push(s(0), push(s(s(0)), empty)) .

-- reduce in STACK-OF-NAT : top (pop push(s(0),push(s(s(0)),empty)))
s(s(0)) : NzNat
(0.000 sec for parse, 2 rewrites(0.000 sec), 2 matches)
STACK-OF-NAT>
```

A meaningless term, as before, remains meaningless.

```
STACK-OF-NAT> parse top pop push(s(0), empty) .

(top (pop push(s(0),empty))) : ?Nat
STACK-OF-NAT> reduce top pop push(s(0), empty) .

-- reduce in STACK-OF-NAT : top (pop push(s(0),empty))
top empty : ?Nat
(0.010 sec for parse, 1 rewrites(0.000 sec), 1 matches)

STACK-OF-NAT>
```

6.6.3 Handling Errors as Errors

The above definition of natural number stacks does not say much about what to do with errors. It simply demarcates the error boundary. If you want to be more specific about handling errors, a standard way is to declare specific error elements (constants), and equalise errors to them. In case of the stacks, the code would look something like

```
...
op * : -> ?Nat
op * : -> ?Stack
...
eq top empty = * .
eq pop empty = * .
...
```

where `*` is meant to be a bottom element. A more elaborate way is to declare such constants as

```
op zero-divisor : -> ?Nat
op top-of-empty-stack : -> ?Nat
```

They are intended to be error messages.

An error sort is automatically declared for each connected component of the sort graph, and has an internal name that starts with "?". In the examples of factorials and stacks, the names derived from single sorts, as `?Rat` and `?Nat`. In general, an error sort may have a complicated name like

```
?SortA+SortB+SortC
```

The "summands" of this name are maximal sorts in a relevant connected component. In making explicit references, you may use a more compact notation: just put a question mark before a sort name in connected components. For example, if you declared

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

the error sort name is internally `?B+C`, but may be referred to as either `?A`, `?B`, or `?C`.

Warning Unlike the former versions, the current version admits error sorts, such as `?Nat`, in operator declarations. Since names of error sorts — in fact, their existence at all — are implementation dependent, you should avoid mentioning error sorts unless absolutely necessary.

Chapter 7

Operator Attributes

Some properties of operators, such as associativity, can be stated as axioms, but if they are so common, it is convenient to be able to use shorthand notations. Moreover, some properties, such as commutativity, are better *not* to be given by axioms, since, if regarded as rewrite rules, such axioms violate desirable properties of the TRS. CafeOBJ has a construct for endowing operators with common properties. Such properties are called *operator attributes* . Some of them affect denotational semantics; others affect only parsing; yet others concern solely with evaluation.

Operator attributes are specified with an optional construct of operator declarations that were explained in Section 3.2.

Syntax 56: *operator declaration with attributes* `op` *operator_symbol* ":" *list_of_sort_names* "->" *sort_name* "{" *attribute_list* "}"

where operator symbols et al. are as before (Section 3.2). We delayed the introduction of this construct, since it is better explained after evaluation commands.

An attribute list is a list of operator attributes separated by blanks. An attribute is either (1) an *equational theory attribute*, which defines an equivalence relation among terms, (2) a *parsing attribute*, which specifies connectivity, (3) an *evaluation attribute*, which controls the evaluation procedure, or (4) neither of these.

Operator declarations introduced by `ops`, `bop`, `bops`, `pred` may also have this optional construct.

7.1 Equational Theory Attribute

CafeOBJ admits four equational theory attributes, as follows. For an infix operator $+$,

(1) Associativity, or $(x + y) + z = x + (y + z)$ for any x, y, and z.
Specified with `associative` or `assoc`.

(2) Commutativity, or $x + y = y + x$ for any x and y.
Specified with `commutative` or `comm`.

(3) Idempotency, or $x + x = x$ for any x.
Specified with `idempotent` or `idem`.

(4) Existence of the identity (here written 0), or $0 + x = x$ and $x + 0 = x$ for any x.
Specified with "`id: 0`".

7.1.1 Associativity

We proceed to a couple of examples for futher illustration. The conjunction "_and_" is associative. In CafeOBJ, it is written as

```
op _and_ : Bool Bool -> Bool { assoc }
```

which means

- No ambiguity arises with or without parentheses. For example,

  ```
  true and false and true
  ```

  has two parses

  ```
  (true and false) and true
  true and (false and true)
  ```

  but by associativity, they are equal.

- During evaluation, terms are matched modulo associativity. For example, in evaluating "`(true and false) and true`" in the presence of

  ```
  eq true and X:Bool = X .
  ```

  the given term does not match the lefthand side as it is, since the first argument is not `true`. But by associativity, the term equals "`true and (false and true)`", and this equivalent term does match the lefthand side. Matching modulo associativity is a procedure to find a matching substitution under arbitrary arrangement of parentheses.

The built-in equality operator "_==_" (Section 4.4.1) takes care of associativity. For example,

```
true and (false and true) == (true and false) and true
```

evaluates to `true`.

`assoc` makes sense only when the operator is binary, with arity of the form "`A A`" and coarity `C` such that `C < A`.

Warning Formerly, CafeOBJ admitted the cases when the arity consists of different sorts. You are warned if the above stricter condition is violated.

Warning The system assumes an operator with `assoc` attribute to be right associative (Section 7.2), unless declared otherwise.

7.1.2 Commutativity

The conjunction is also commutative. In CafeOBJ, it is written as

```
op _and_ : Bool Bool -> Bool { assoc comm }
```

Commutativity may be specified with the equation

```
eq X:Bool and Y:Bool = Y and X .
```

which is fine as an equation. As a rewrite rule, however, it is diabolically problematic, since it generates infinite rewrite sequences. For this reason, commutativity should be stated by operator attribute.

Like `assoc`, `comm` affects evaluation. For example, the term "`false and true`" does not match the lefthand side of

```
eq X:Bool and false = false .
```

but an equivalent term `true and false` does.

"`_==_`" takes care of commutativity as well as associativity, so

```
true and (false and true) == (true and true) and false
```

evaluates to `true`.

`comm` makes sense only when the operator is binary and its arity sorts are the same.

Warning Formerly, CafeOBJ admitted the cases when the arity sorts have a common supersort.

7.1.3 Identity

"`_and_`" has `true` as (left and right) identity, which is written in CafeOBJ as

```
op _and_ : Bool Bool -> Bool { assoc comm id: true }
```

The attribute "`id: true`" is equivalent to the following equations.

```
eq true and X:Bool = X .
eq X:Bool and true = X .
```

(In fact, since `comm` is also given, one of the above is enough.) Identity attributes affect evaluation. Under the above operator declaration, `false and true` is, *without* evaluation, identified with `false`.

"`id:`" makes sense only when the operator is binary, the arity sorts and the coarity sort are the same, and the identity term belongs to that sort.

Warning Formerly, CafeOBJ admitted the cases when the coarity sort is different.

Warning The current implementation of CafeOBJ applies a completion procedure to realise matching modulo identity. The procedure often generates a great many of equations, leading to severe performace degeneration, or even to non-termination. To prohibit unwanted completion, use "`idr:`", instead of "`id:`". This attribute has the the same meaning as "`id:`".

7.1.4 Idempotency

That "`_and_`" is idempotent is stated as follows.

```
op _and_ : Bool Bool -> Bool { assoc comm id: true idem }
```

`idem` is equivalent to the equation

```
eq X:Bool and X = X .
```

Idempotency affects evaluation. For example, "`true and true`" and `true` matches modulo idempotency.

`idem` makes sense only when the operator is binary and the arity sorts and the coarity sort are the same.

Warning Formerly there was no explicit condition.

Warning In the current implementation, an equation like the above is actually added. The system sometimes misbehaves; the only certainty about idempotency is that matching modulo the *combination* of associativity, commutativity, and idempotency works correctly.

7.1.5 Inheriting Equational Theory Attributes

Equational theory attributes are inherited by operators, if they have the same name and are declared on lower sorts. For example, when you declare

```
[ Nat < Int ]
op _+_ : Nat Nat -> Nat
op _+_ : Int Int -> Int { assoc }
```

the operator "`_+_`" on `Nat` is assumed associative. Since a subsort is a subset, such attribute inheritance is quite natural: if associativity holds for all the integers, it a forteriori holds for all the natural numbers. As a special case, an identity is inherited only when the identity term belongs to the lower sort.

7.2 Parsing Attribute

Parsing attributes help disambiguate parsing. They are either

(1) Precedence, specified by "`prec:` integer", or

(2) Left or right associativity, specified by `l-assoc` or `r-assoc`, respectively.

7.2.1 Precedence

A precedence is an integer, and lesser precedences indicate stronger connectivity. For example,

```
op _+_ : Int Int -> Int { prec: 33 }
op _*_ : Int Int -> Int { prec: 31 }
```

states that "`_*_`" connects stronger than "`_+_`", and "`1 + 2 * 3`" parses as "`1 + (2 * 3)`".

The default precedences depend on term constructs.

- Standard operators have precedence 0. Hence the term of the form `f(a)` connects strongest.
- When an operator is mixfix,
  - If the operator symbol does not start or end with "`_`", that is, no argument appears outside the term construct, the precedence is zero. An example is a singleton operator "`{_}`";
  - if it is a prefix unary operator, the precedence is 15. The unary "`-_`" offers an example;
  - Otherwise, the precedence is 41.

Let us show examples.

```
CafeOBJ> module PREC-TEST {
  [ S ]
  op _+_ : S S -> S { prec: 33 }
  op _*_ : S S -> S { prec: 31 }
  op -_ : S -> S
  }

-- defining module PREC-TEST...._* done.
CafeOBJ> select PREC-TEST

PREC-TEST> parse X:S + Y:S * Z:S .

(X + (Y * Z)) : S
PREC-TEST> parse - X:S + Y:S .

((- X) + Y) : S
PREC-TEST>
```

According to the default rules, "`-_`" has precedence 15, so connects stronger than "`_+_`". Hence the result of the second parse.

> **Warning** The current implementation accepts precedences between 0 and 127 (inclusive). Out-of-bound precedences cause warnings, and are ignored.

7.2.2 Left/Right Associativity

`l-assoc` states that the operator, which must be binary, associates to the left, and `r-assoc` to the right. A typical usage is as follows. Under the declaration

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

"`X:S + X + X`" has two parses

```
(X + X) + X
X + (X + X)
```

If you declare

```
op _+_ : S S -> S { l-assoc }
```

the parse is uniquely determined as "`(X + X) + X`", while

```
op _+_ : S S -> S { r-assoc }
```

dictates the parse to be "`X + (X + X)`".

> **Warning** When equational-theoretic associativity `assoc` is declared, the system has to decide how to decide connectivity. In that case, "`X + (X + X)`" is equal to "`(X + X) + X`" anyway, but it is necessary to decide on one or the other, to store and manipulate terms. In the current implementation, the system assumes right associativity.

Left/right associativity is allowed only on operators whose arities are both supersorts of the coarity.

> **Warning** Note that the conditions for `assoc` and for `l-assoc`, `r-assoc` differ. `l-assoc` or `r-assoc` is a parsing instruction. There is no inherent reason to prohibit these attributes, as long as you feel like it. In fact, the above condition only states that, if it does not hold, the attribute is almost meaningless.

7.3 Constructor Attribute

The attribute `constr` states that the operator is a constructor of the coarity sort. For example,

```
op nil : -> List { constr }
op __ : List List -> List { constr }
```

states that `nil` and `__` are constructors of `List`. The idea of constructors is very important for many purposes, especially for theorem proving. The current system, however, does not take advantage of this concept in any way. To the system, `constr` is a comment.

7.4 Evaluation Strategy

CafeOBJ allows the user to specify evaluation strategies of terms. In general, evaluation is nondeterministic. In selecting which rule to apply to which subterm, there is a certain license. And the selection affects performance. It may even decide whether the evaluation terminates.

7.4.1 E-Strategy Attribute

Many recent functional programming languages are using lazy evaluation strategies. There are several arguments for this trend, but unfettered laziness is not recommended, like in everyday life. One way to avoid its harmful effects is to incorporate in compilers a mechanism to intermingle eager evaluation strategies, so long as it does not change the results of computation. To see if it affects the results, strictness analyses are employed.

In CafeOBJ, you need not be thoroughly lazy or eager. It allows you to assign different strategies to different operators. This means that you can have the best of both worlds, or eat and have a cake at once. Moreover, the system computes default strategies case by case, based on a rapid strictness analysis.

In accordance to a certain folklore, we sometimes say *E-strategy* for evaluation strategy. E-strategies are operator attributes, and are written in this form:

Syntax 57: *evaluation strategy*
`strat:` "(" *integer_list* ")"

An integer list is a list of integers separated by blanks. Each integer in the list represents an argument. 1 is the first argument, 2 is the second, and so on, and 0 is the whole term.

E-strategies determine the order of evaluation. As an informative example, a built-in conditional is declared as follows.

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

This E-stragety ordains that the first argument (conditional part) be evaluated first, then the whole argument. Notice that the second and third arguments (`then` and `else` parts) do not appear in the list. These arguments are not evaluated, unless the whole term is evaluated and they are no longer arguments. Depending on the result of evaluating the first argument, one of these arguments is entirely irrelevant to the final result, so this strategy is clearly reasonable.

Another example is an eager strategy.

```
op _+_ : Int Int -> Int { strat: (1 2 0) }
```

The E-strategy requires the two arguments be evaluated before the whole term.

7.4.2 Default Strategy

If no explicit E-strategy is given, the default strategy is computed as follows. For each argument,

> Evaluate the argument before the whole term, if there is a rewrite rule that defines the operator such that, in the place of the argument, a non-variable term appears.

For example, given a binary operator `f` and suitable variables `X` and `Y`, in the place of the first argument is a variable in

```
eq f(X,Y) = h(X) .
```

while it does not in

```
eq f(g(X),Y) = h(Y) .
```

If all the rewrite rules are of the former type, the first argument need not be evaluated immediately. Since any term (of a relevant sort) matches a variable, matching possibilities are unaffected. Furthermore, if the only rewrite rule defining `f` is

```
eq f(X,Y) = h(Y) .
```

you are well advised to evaluate it later (or never): the righthand side does not contain `X`, so evaluating it is futile.

On the other hand, if there are rules that contain non-variable terms in the argument place, the result of evaluating the argument affects matchability, so it is safe to evaluate it first.

Warning To be on the safe side, the default strategy always ends with `0`. Former versions added last zeros even for user-supplied stategies. We concluded that this practice reminded us of overprotective nannies, and abandoned it.

If an operator definition is recursive, CafeOBJ may ignore an explicit E-strategy, and apply recursive rules successively. This is a kind of optimisation. To avoid this behaviour, give an E-strategy that does not have `0` at the head of the list.

7.4.3 Lazy Evaluation

CafeOBJ provides two ways to specify lazy evaluation.

a. Omit the arguments to be evaluated later, or

b. Put the corresponding negative integers.

Delayed `cons` is thus declared as either

```
op cons : Sexp Sexp -> Sexp { strat: (0) } -- (a)
```

or

```
op cons : Sexp Sexp -> Sexpt { strat: (-1 -2) } -- (b)
```

With the former declaration, both arguments are not evaluated unless they get outside of `cons`. For example, suppose `car` is defined as

```
op car : Sexp -> Sexp { strat: (1 0) }
vars X Y : Sexp
eq car(cons(X,Y)) = X . -- *
```

and you try to reduce the term

```
car(cons(1 + 0, nil))
```

where `nil` is the usual constant. According to the given E-strategies,

1. The argument of `car` is to be evaluated first.
2. But the argument is dominated by `cons`, whose arguments are not to be evaluated, since `1`, `2` are not in the E-strategy. So "`cons(1 + 0, nil)`" is returned as it is.
3. The evaluation for `car` continues, and the next argument in the E-strategy — in fact `0`, the whole term — is to be evaluated.
4. Using the equation marked `*`, the whole term is rewritten to "`1 + 0`".
5. Evaluation continues with this term.

With the declaration `(b)`, both arguments are not evaluated until so forced. Evaluation is forced, when the argument is involved in matching. For example, to see the applicability of the equation (rewrite rule)

```
eq cadr(cons(X,cons(Y,Z))) = Y .
```

it is necessary to know whether the second argument of the outer `cons` is dominated by `cons`. Suppose a term has been rewritten so far to

```
cadr(cons(1,cdr(cons(2,cons(3,nil)))))
```

This term does not match the lhs as it is, because the second argument of the outmost `cons` is dominated by `cdr`, not by `cons`. It does match, however, if the second argument is reduced (with the usual rule for `cdr`) to `cons(3,nil)`. `-2` in the strategy states that the second argument is to be evaluated just at such moments.

Negative integers cannot be used if the operator is declared associative or commutative. In such cases, reduction "on demand", as explained above, is nigh impossible to implement.

7.4.4 Interpreting E-Strategies

To understand the exact meaning of E-strategies, it is instructive to know the overall algorithm of evaluation. It is shown below. Remember that an E-strategy is a list of integers.

Given a term t,

1. Let l be the E-strategy of the dominant operator of t.
2. If l is empty, exit with t as result. Otherwise, let n be the first element of l.

   2-1. If $n = 0$, search a rewrite rule applicable to the whole t. If one such is found, replace t with the substitution instance t', and go back to 1.

   2-2. If $n > 0$, call this procedure recursively with the n-th argument of t as input. Replace the argument with the result, and proceed to 3.

   2-3. If $n < 0$, mark the $-n$-th argument of t as "on demand", and proceed to 3.

3. Remove the first element from l and go back to 2.

Note that, after the step 2-1., the control is back to 1., and the rest of the current strategy is discarded. Note also that, when a negative integer appears, the corresponding argument is just marked and nothing else is done. The mark reveals its effects only during matching.

7.5 Memo

To evaluate similar terms repetitively is tedious to humans, but acceptable to the system. To evaluate *the same* terms repetitively, however, is sometimes too much, even to the system. It silently complains, by taking long time in responding. For the sake of efficiency, it helps if you state in advance that such and such evaluations are likely to occur quite often, and that remembering the previous results is a good idea.

Hence an attribute, called *memo*. This operator attribute tells the system to remember the results of evaluation where the operator appeared. More specifically, if you declare

```
op f**k : S S S -> S { memo }
```

and evaluate a term containing `f**k(t1,t2,t3)`, the system take note of (1) `f**k(t1,t2,t3)` itself, (2) `f**k(r1,r2,r3)`, where `ri`'s are the results of evaluating `ti`'s, and (3) the result of evaluating (1). If the need arises to evaluate (1) again, the system gets the result (3) immediately. This mechanism not only speeds up the computation, it also saves memory space, when you evaluate terms that contain common subterms.

The memo attribute can only specified on functions, i.e. operators not defined by rules (See for rules).

The system does or does not remember the evaluation results after each evaluation. It is controlled by a switch, and

```
CafeOBJ> set clean memo on
```

tell the system to be forgetful. The default is `off`. It is also possible to initialise memo storages by the following command.

```
CafeOBJ> clean memo
```

The memo attributes may be ignored if you turn a switch off, as

```
CafeOBJ> set memo off
```

7.5.1 Coherence

As explained in Section 6.2.1 and 6.2.4, in evaluating a term, application of behavioural axioms is prohibited unless the redex is enclosed in an observation, and this restriction is to ensure the soundness of evaluation. This implies that the system can ignore the restriction so long as the application does not lead to an incorrect reasoning.

An attribute called `coherent` may be used to tell the system to loosen the guard. It asserts that the operator acts indistinguishably on behaviourally equivalent terms, so that it is in effect a behavioural operator. For example, the declarations

```
[ T ]
*[ S ]*
ops a b : -> S
```

```
bop g : S -> T
op f : S -> T { coherent }
beq [ a-sim-b ] a = b .
```

allow the system to apply `a-sim-b` to `f(a)`, where a is not guarded by an observation, as well as to `g(a)`, where a is.

7.6 Operator Attributes and Evaluations

Let us see how various attributes affect evaluation. In the case of an associative operator with identity,

```
module AZ-OP {
 [ S ]
 ops : a b c phi : -> S
 op __ : S S -> S { associative id: phi }
}
```

you may compute as follows.

```
phi a phi b phi c ->
a phi b phi c ->
a b phi c ->
a b c
```

where `phi`'s are eliminated wherever they appear. Indeed the system computes exactly like you.

```
CafeOBJ> select AZ-OP

AZ-OP> set trace on

AZ-OP> reduce phi a phi b phi c .

-- reduce in AZ-OP : phi a phi b phi c
1>[1] rule: eq [ident12] : phi X-ID:S = X-ID:S
    { X-ID:S |-> a phi b phi c }
1<[1] phi a phi b phi c --> a phi b phi c
1>[2] rule: eq A2 phi X-ID:S A1 = A2 X-ID:S A1
    { A1 |-> phi c, X-ID:S |-> b, A2 |-> a }
1<[2] a phi b phi c --> a b phi c
1>[3] rule: eq A1 phi X-ID:S = A1 X-ID:S
    { X-ID:S |-> c, A1 |-> a b }
1<[3] a b phi c --> a b c
a b c : S
(0.000 sec for parse, 3 rewrites(0.260 sec), 13 matches)
AZ-OP>
```

The equations in this trace were generated by the system, to cope with identity. The exact mechanism need not be your concern, but beware that these extra equations may cause havoc (Section 7.8).

If you inspect the statistics, it is apparent that the numbers of matching attempts greatly exceeded those of rewrites. This is a general fact about evaluation modulo equational theory.

See what happens if the operator is also commutative, as in

```
AZ-OP> module ACZ-OP {
  [ S ]
  ops : a b c phi : -> S
  op __ : S S -> S { associative commutative id: phi }
}

-- defining module ACZ-OP......_.* done.
AZ-OP> select ACZ-OP

ACZ-OP> reduce phi a phi b phi c .

-- reduce in ACZ-OP : phi a phi b phi c
1>[1] rule: eq [ident14] : phi X-ID:S = X-ID:S
    { X-ID:S |-> a b phi phi c }
1<[1] phi a phi b phi c --> a b phi phi c
1>[2] rule: eq [ident14] : phi X-ID:S = X-ID:S
    { X-ID:S |-> a b phi c }
1<[2] a b phi phi c --> a b phi c
1>[3] rule: eq [ident14] : phi X-ID:S = X-ID:S
    { X-ID:S |-> a b c }
1<[3] a b phi c --> a b c
a b c : S
(0.010 sec for parse, 3 rewrites(0.010 sec), 5 matches)
ACZ-OP>
```

Again phi's were eliminated, but different rewrite rules were used, and, relying on commutativity, the system changed the order of listing at the first step.

Adding idempotency, we get

```
ACI-OP> module ACIZ-OP {
  [ S ]
  ops a b c phi : -> S
  op __ : S S -> S { assoc comm idem id: phi }
}

-- defining module ACIZ......_.* done.
ACI-OP> reduce in ACIZ-OP : phi a phi b phi c .

-- reduce in ACIZ-OP : phi a phi b phi c
1>[1] rule: eq AC U-idem:S U-idem:S = AC U-idem:S
    { AC |-> phi a b c, U-idem:S |-> phi }
1<[1] phi a phi b phi c --> phi a b c
1>[2] rule: eq [ident16] : phi X-ID:S = X-ID:S
    { X-ID:S |-> a b c }
1<[2] phi a b c --> a b c
a b c : S
(0.000 sec for parse, 2 rewrites(0.000 sec), 9 matches)
ACI-OP>
```

phi's were stripped, but the reason for the first step (when two phi's were eliminated at once) was not that phi was the identity, but that "__" was idempotent.

7.7 Further Example: Propositional Calculus

The next example is a decision procedure of propositional logic. In this example, rewriting modulo associativity and commutativity is decisive.

CafeOBJ has a built-in module PROPC, which is declared as follows.

```
  CafeOBJ> show PROPC
  sys:module! PROPC
        principal-sort Prop
 {
    imports {
      protecting (TRUTH)
    }
    signature {
      [ Prop, Identifier Bool < Prop ]
      op _ and _ : Prop Prop -> Prop { assoc comm constr prec: 55 r-assoc }
      op _ xor _ : Prop Prop -> Prop { assoc comm constr prec: 57 r-assoc }
      op _ or _ : Prop Prop -> Prop { assoc comm prec: 59 r-assoc }
      op not _ : Prop -> Prop { strat: (0 1) prec: 53 }
      op _ -> _ : Prop Prop -> Prop { strat: (0 1 2) prec: 61 }
      op _ <-> _ : Prop Prop -> Prop { assoc prec: 63 r-assoc }
    }
    axioms {
      var p : Prop
      var q : Prop
      var r : Prop
      eq p and false = false .
      eq p and true = p .
      eq p and p = p .
      eq p xor false = p .
      eq p xor p = false .
      eq p and (q xor r) = p and q xor p and r .
      eq p or q = p and q xor p xor q .
      eq not p = p xor true .
      eq p -> q = p and q xor p xor true .
      eq p <-> q = p xor q xor true .
    }
  }

  CafeOBJ>
```

"`sys:module!`" is an introductory keyword for built-in modules.

- An atomic proposition is an element of the built-in sort `Identifier`, declared in `CHAOS:IDENTIFIER`, or of the sort `Bool`, declared in `TRUTH`. `Identifier` contains character strings. `Bool` contains `true` and `false`.
- Propositional connectives are `and`, `or`, `->`, `not`, `<->`, and `xor` (exclusive or).
- An irreducible term is constructed out of `and`, `xor`, `true`, `false`, and identifiers. It is in disjunctive normal form (by regarding `xor` as disjunction). In particular, tautologies are reducible to `true`.
- The precedences of connectives are ordered as `not`, `and`, `xor`, `or`, `->`, `<->`.

Evaluations in this module proceed as follows. Firstly,

```
  a -> b <-> not b -> not a
```

states $a \to b \leftrightarrow \neg b \to \neg a$, a tautology (equivalence of contrapositives).

```
CafeOBJ> select PROPC

PROPC> set trace on

PROPC> reduce a -> b <-> not b -> not a .

-- reduce in PROPC : a -> b <-> not b -> not a
1>[1] rule: eq p:Prop -> q:Prop = p:Prop and q:Prop xor p:Prop xor
       true
    { p:Prop |-> a, q:Prop |-> b }
1<[1] a -> b --> a and b xor a xor true
1>[2] rule: eq p:Prop -> q:Prop = p:Prop and q:Prop xor p:Prop xor
       true
    { p:Prop |-> not b, q:Prop |-> not a }
1<[2] not b -> not a --> not b and not a xor not b xor true
1>[3] rule: eq not p:Prop = p:Prop xor true
    { p:Prop |-> b }
1<[3] not b --> b xor true
1>[4] rule: eq not p:Prop = p:Prop xor true
    { p:Prop |-> a }
1<[4] not a --> a xor true
1>[5] rule: eq p:Prop and (q:Prop xor r:Prop) = p:Prop and q:Prop
      xor p:Prop and r:Prop
    { p:Prop |-> b xor true, q:Prop |-> true, r:Prop |-> a }
1<[5] (b xor true) and (a xor true) --> (b xor true) and true xor
     (b xor true) and a
1>[6] rule: eq p:Prop and true = p:Prop
    { p:Prop |-> b xor true }
1<[6] (b xor true) and true --> b xor true
1>[7] rule: eq p:Prop and (q:Prop xor r:Prop) = p:Prop and q:Prop
      xor p:Prop and r:Prop
    { p:Prop |-> a, q:Prop |-> true, r:Prop |-> b }
1<[7] (b xor true) and a --> a and true xor a and b
1>[8] rule: eq p:Prop and true = p:Prop
    { p:Prop |-> a }
1<[8] a and true --> a
1>[9] rule: eq AC xor p:Prop xor p:Prop = AC xor false
    { AC |-> true xor a xor a and b, p:Prop |-> true xor b }
1<[9] b xor true xor a xor a and b xor b xor true xor true --> true
    xor a xor a and b xor false
1>[10] rule: eq p:Prop xor false = p:Prop
    { p:Prop |-> a and b xor true xor a }
1<[10] true xor a xor a and b xor false --> a and b xor true xor
    a
1>[11] rule: eq p:Prop <-> q:Prop = p:Prop xor q:Prop xor true
    { p:Prop |-> a and b xor a xor true, q:Prop |-> a and b xor true
       xor a }
1<[11] a and b xor a xor true <-> a and b xor true xor a --> a and
     b xor a xor true xor a and b xor true xor a xor true
1>[12] rule: eq AC xor p:Prop xor p:Prop = AC xor false
    { AC |-> true, p:Prop |-> true xor a and b xor a }
1<[12] a and b xor a xor true xor a and b xor true xor a xor true
     --> true xor false
1>[13] rule: eq p:Prop xor false = p:Prop
    { p:Prop |-> true }
```

```
1<[13] true xor false --> true
true : Bool
(0.000 sec for parse, 13 rewrites(0.140 sec), 75 matches)
PROPC>
```

Observe that associativity and commutativity are frequently used. To trace everything makes the exposition rather long, so we omit traces in the sequel.

```
PROPC> set trace off

PROPC>  reduce not(a or b) <-> not a and not b .

-- reduce in PROPC : not (a or b) <-> not a and not b
true : Bool
(0.000 sec for parse, 11 rewrites(0.020 sec), 70 matches)
PROPC> reduce c or c and d <-> c .

-- reduce in PROPC : c or c and d <-> c
true : Bool
(0.000 sec for parse, 7 rewrites(0.190 sec), 34 matches)
PROPC> reduce a <-> not c .

-- reduce in PROPC : a <-> not c
a xor c : Prop
(0.000 sec for parse, 4 rewrites(0.010 sec), 15 matches)
PROPC> reduce a and b xor c xor b and a .

-- reduce in PROPC : a and b xor c xor b and a
c : Identifier
(0.010 sec for parse, 2 rewrites(0.000 sec), 21 matches)
PROPC> reduce a <-> a <-> a <-> a .

-- reduce in PROPC : a <-> a <-> a <-> a
true : Bool
(0.010 sec for parse, 9 rewrites(0.000 sec), 18 matches)
PROPC> reduce a -> b and c <-> (a -> b) and (a -> c) .

-- reduce in PROPC : a -> b and c <-> (a -> b) and (a -> c)
true : Bool
(0.010 sec for parse, 25 rewrites(0.050 sec), 184 matches)
PROPC>
```

It is pleasing to see that the usual logical truths are proved to be true, by just evaluating the corresponding terms. Here the logical basis of the language is in stark relief. If you think of rewrite rules as deduction rules, a computation is a proof. The set of rewrite rules is sound iff all the statements reducible to `true` are valid. It is complete iff all the valid statements are reducible to `true`.

7.8 Limitation of the Implementation

As a cautious conclusion to operator attributes, we must point to the major limitations of the current implementation. As mentioned earlier, the treatment of idempotency is inadequate. The other two problems concern the enormity of extra rewrite rules. You should have noticed that, to realise rewriting modulo identity or associativity, the system adds equations surreptitiously. Consider first the case of identities.

Suppose in `AZ-OP` in Section 7.6, there were another operator declaration

```
op first : S -> S
```

and an equation

```
eq first(X:S X':S) = X .
```

A special case of the equation is

```
eq first(X) = X .
```

where `X'` is bound to `phi` and the identity is cancelled. The completion procedure for implementing identities does add this equation, as well as

```
eq first(X') = phi .
```

which is a specialisation to the case when `X` is `phi`. Thus we get a specification where

```
phi = first(a) = a = first(a b) = phi = ...
```

holds. This result is somewhat unexpected. As a TRS, the two equations annihilate the Church-Rosser property.

A different problem arises if you modify the declaration of `xor` in `PROPC` (Section 7.7) as

```
op _xor_ : Prop Prop -> Prop { assoc comm id: false }
```

i.e., adding the identity attribute. Then in the presence of the equation

```
eq p and (q xor r) = p and q xor p and r .
```

the completion procedure adds

```
eq p and r = (p and false) xor (p and r) .
```

which is a special case, where `q` is replaced by `false`. This equation is hazardous, since you now get an infinite rewrite sequence

```
a and b -->
(a and false) xor (a and b) -->
(a and false) xor (a and false) xor (a and b) -->
    :
```

Even if such a hazard is avoided, performance may be compromised, by hundreds of extra unwanted equations. If you encounter misbehaviours, then, "`id:`" is a possible suspect. If it is the culprit, change to "`idr:`", which does not force completion.

As to associativity, a fateful combination with sort ordering may generate dozens of unnecessary equations. For example, in the built-in module `INT`, `Nat` is a subsort of `Int`, and the addition "`_+_`" is declared on both sorts. As a consequence, an equation

```
eq 0 + I = I .
```

induces an extra equation

```
eq 0 + I + ac_E = ac_E + I .
```

This equation has a lefthand side that is costly to match, yet adds nothing to the power of evaluation. To check the necessity of such extra equations is not easy. As of now, the system bets on the safer side, and adds equations nonchalantly.

Chapter 8

Module Structure

8.1 Names and Contexts

8.1.1 Context and Current Module

Import declarations define a hierarchy of modules. As a directed graph, it is acyclic: a module cannot import itself, either directly or indirectly. The directed acyclic graph is called *context*, and for a paricular module `M` in the graph, the complete subgraph reachable from `M` is called the context of `M`. You may regard the context of `M` as collections of sorts, operators, variables, and axioms, which can be referred to within `M`.

The concept of current module is to establish a context in the above sense. Such commands as `reduce` and `parse` presuppose a context. The current module is set by the following command.

Syntax 58: `select` *command* `select` *module_name*

You are always reminded of the current module by the prompt — you already have seen many session examples.

You may make the context follow your every step, so that if you define a module or inspect its contents, the current module switches to that module, by the command

```
CafeOBJ> set auto context on
```

The default is `off`. The effects of this switch are illustrated in the following session.

```
CafeOBJ> set auto context on

CafeOBJ> module M { [ S ] op e : -> S }

-- defining module M.._* done.
M> module N { [ S ] }

-- defining module N._* done.
N> show M
module M {
  imports {
    protecting (BOOL)
  }
  signature {
    [ S ]
```

```
    op e : -> S
  }
}

M> show N
module N {
  imports {
    protecting (BOOL)
  }
  signature {
    [ S ]
  }
}

N>
```

Notice that the prompt changed to M (N), each time M (N) was defined or inspected. The current module — hence context — can be shown as follows.

```
N> show context
-- current context :
[module] N
[special bindings]
  $$term    = none.
  $$subterm = no subterm selection is made by ‘choose’.
[bindings] empty.
[selections]  empty.
[pending actions]  none.
[stop pattern] not specified.

N>
```

The output contains the states that are local to the current context. For example, under `bindings` are listed context variables (Section 6.5). The others concern theorem proving tools, to be explained (Chapter 9).

`show` command is also used to print contexts. To print the context of `M`, type

```
CafeOBJ> show module tree M
```

A submodule of a module is identified by a name, so during a session it is possible for the definition of a submodule to change — when you redefine that submodule, or when you unintentionally use the same name for a different module. In such a case, the system does not take any housekeeping exercise until necessary — until, for example, an evaluation command is entered. You may make the system to be more diligent, by flicking a switch.

```
CafeOBJ> set auto reconstruct on
```

When the switch is on, every time a module is redefined, every context surrounding it is forced to reflect the change at once.

If you are so inclined, you may change the prompt once and for all.

```
CafeOBJ> prompt Haskell>

Haskell>
```

In the reset of the session, the prompt would not change until you invoke the command again[1].

[1] For an obvious reason, the above change does not force the system to accept Haskell.

8.1.2 Module Sharing

It is possible that some modules are imported several times. For example, the declarations

```
module M0 { ... }
module M00 { protecting (M0) ... }
module M01 { protecting (M0) ... }
module M001 { protecting (M00) protecting (M01) ... }
```

makes `M001` import `M0` twice. The basic convention is that, in such cases, `M0` is regarded as imported only once, and its single copy is shared throughout. This convention works fine when a module is imported with `protecting` mode, since a module then has essentially the same models everywhere. The problem arises when `extending` or `using` modes appear somewhere. In general, importation modes are determined as follows.

Let us write `M -m-> M'` if `M` directly imports `M'` with importation mode `m`. When `M` (directly or indirectly) imports `M'`, there are several chains

```
M -m1-> M1 -m2-> ... -mn-> Mn = M'
```

to `M'`. Each such chain determines the mode with which `M` imports `M'`, as the weakest among all the `mi`'s. For example, in the chain

```
M -pr-> M1 -ex-> M2 -pr-> M'
```

`M'` is imported via `ex` mode.

The mode with which `M'` is imported is the weakest among all the chains along which `M'` is imported.

8.1.3 Qualifying Names

In Section 3.3.2, we have shown how to qualify terms by sort names. You can also qualify sort names and operator names, this time by module names.

CafeOBJ allows you to use the same name for different sorts. A typical case is to declare sorts in different but related modules: variants of stack definitions, for example, are likely to have a common sort name `Stack`. So long as they do not appear in the same context, those sorts do not cause confusion. The problem arises when you import two modules that contain sorts of the same name. It is unlikely to import both modules directly, but it is definitely possible to import them indirectly. To avoid confusion you can qualify names as follows.

Syntax 59: *reference to sort/operator/parameters*
name[" . "*module_name*]

where the name is a sort name, an operator name, or a parameter name (Section 8.2.1). According to the above syntactical rule, the references that appeared so far, such as `Nat`, could be qualified, such as "`Nat.SIMPLE-NAT`". A module name may be a parameter name (Section 8.2.1).

To avoid ambiguity, if an operator name contains blanks or periods, it must be parenthesised, as "`(_._).M`".

The referent of a qualified name is the one that is in the context of the qualifier. "`Nat.SIMPLE-NAT`" refers to the sort `Nat` in the context of `SIMPLE-NAT`. Note that this does *not* imply that `Nat` is declared in `SIMPLE-NAT`: it only says that *somewhere* in the context of `SIMPLE-NAT`, `Nat` is declared. If, in the context of `SIMPLE-NAT`, more than one `Nat` are declared, the qualifier fails to identify which. In that case, you have to be more specific.

Warning The above qualification of operator names are for *referring* to operators in `show` commands, mappings, etc. You cannot use qualification in writing a term.

8.1.4 Referring to Operators

Operator names appear in renaming and view declarations (both are to be explained). In these constructs, a single name may refer to more than one operators, due to overloading. Special care needs to be taken, therefore, to see which operator names refer to which operators.

Note first that you have to distinguish operators with different number of arguments, as

```
op f : Nat -> Nat
```

and

```
op f : Nat Nat -> Nat
```

This problem does not arise if operators are mixfix, since their symbols contain placeholders "_"s. All in all, you have this definition.

Syntax 60: *operator name*
mixfix_operator_symbol | *standard_operator_symbol*["/" *number_of_arguments*]

You should not insert blanks between the operator symbol and "/", or between "/" and the number of arguments. Using this form, the above two declarations may be distinguished as "`f/1`" and "`f/2`".

With this preparation, let us define in detail the relationship between operator names and their referents. In CafeOBJ, an operator name refers to a group of operators, rather than a single operator. Given a module `M`, consider, for example, an operator declaration

```
op f : Nat Nat -> Nat
```

in `M`, and let G be the group this operator belongs to.

1. Another operator with the same name (i.e. "`f/2`"), and whose arity is in the same connected component as `Nat Nat`, belongs to G.
2. Yet another operator, with the same name but with arity outside the connected component, belongs to G only if it is declared in `M`.
3. No other operator belongs to G.

Here the connected component of "`Nat Nat`" is taken from the product graph. The operator name "`f/2`" in the context of `M` refers to the group G, which may contain several distinct operators, due to the above rules. Accordingly, the qualified name "`(f/2).M`" refers to G.

Intuitively, qualification by `M` states that the name is declared in `M`. Hence 2. precludes operators declared in other modules. But what about 1.? Consider, for example, the code

```
module NAT {
  [ Nat ]
  ...
  op _+_ : Nat Nat -> Nat
  ...
}

module INT {
  protecting (NAT)
  [ Nat < Int ]
  ...
  op _+_ : Int Int -> Int
  ...
}
```

and the reference "(_+_).INT". It certainly refers to "_+_" on Int, but it should also refer to "_+_" on Nat, since CafeOBJ regards an overloaded operator on subsorts as the restriction. On Nat, two "_+_"s must behave identically. Hence the rule 1.

8.2 Parameters and Views

One of the salient features of CafeOBJ is a parameter mechanism. By putting aside certain sorts and operators as parameters, you can declare a generic module. A parameter unit is itself a module so that, on top of syntactical requirements, semantic constraints may be imposed on actual parameters. To instantiate a parameterised module is to replace the parameters with modules that satisfy the syntactic *and* semantic constraints.

8.2.1 Parameterised Module

To declare a parameterised module, attach a parameter interface to the module declaration construct.

Syntax 61: *parameterised module*
  module *module_name* "(" *list_of_parameters* ")" "{"
    module_element *
  "}"

A list of parameters is comma-separated. A *parameter* has the following construct.

Syntax 62: *parameter*
  [*importation_mode*] *parameter_name* "::" *module_name*

A parameter name is an arbitrary character string, and is used to refer to the parameter module in the module body. The module name after "::" describes requirements that must be satisfied by the actual parameters. The same module may act as distinct parameters, and parameter names distinguish them. For importation modes, which are optional, see Section 8.3.2.

Warning In ancient times, parameters were enclosed with square brackets "[", "]".

Warning Do not forget to put blanks before and after "::".

For example, the following modules define indexed lists, where both indices and data may come from arbitrary sets.

```
module* ONE { [ Elt ] }

module! ILIST ( IDX :: ONE, DAT :: ONE ) {
  [ Ilist ]
  [ Elt.DAT < ErrD ]
  op undef : -> ErrD
  op empty : -> Ilist
  op put : Elt.IDX Elt.DAT Ilist -> Ilist
  op _[_] : Ilist Elt.IDX -> ErrD
  -- ----------------------------
  vars I I' : Elt.IDX
  var D : Elt.DAT
  var L : Ilist
  eq put(I,D,L) [ I' ] = if I == I' then D else L [ I' ] fi .
  eq empty [ I ] = undef .
}
```

Observe how the same module ONE is used as different parameters. Inside ILIST, sorts and operators of a parameter may be referred to with qualified names, like "`Elt.DAT`" (cf. Section 8.1.3).

As an aside, the current version of CafeOBJ has a built-in module called `TRIV`, which is the same as `ONE` above. Using this built-in, you may as well have written

```
module! ILIST ( IDX :: TRIV, DAT :: TRIV ) { ...
```

Further, error sorts, such as `ErrD`, are implicitly declared (Section 6.6), so you need not declare `ErrD` as above.

8.2.2 Pre-view (Not Preview)

This section contains a preliminary to the following sections, and is rather technical. To bind actual parameter modules to formals, we need ways to relate two modules. Since modules denote structures, we rely on the idea of structure-preserving mappings. Such mappings are usually called homomorphisms. Intuitively, in

```
module M { [ S ] }
module M' { [ S' ] }
module N { [ T U ] }
```

the modules `M` and `M'` are essentially the same, since the difference lies only in the naming; and `M` (or `M'`) may be embedded in `N`, since `N` defines an enlarged (two-sorted) structure. The following definitions are nothing more than the statements of the obvious, like this tiny example. They are complicated only because CafeOBJ is founded on a rich logical base.

Derived Operators A derived operator is just a term regarded as an operator. For example, under the module `MULT` (Section 6.3.1) and for variables `X` and `Y` of sort `Nat`, `s(0)`, `s(s(X))`, "`s(0) + X`", "`s(X) * s(Y)`", and "`X * X`" are all terms, hence derived operators. The important point here is that each such term can be assigned a rank, inductively. We have

term	rank
`s(0)`	`-> NzNat`
`s(s(X))`	`Nat -> NzNat`
`s(0) + X`	`Nat -> NzNat`
`s(X) * s(Y)`	`Nat Nat -> Nat`
`X * X`	`Nat -> Nat`

It is easy[2] to calculate the rank of a derived operator: the sorts of the variables constitute the arity, and the coarity of the dominant operator is the coarity.

Further, given a signature, the derived signature is the one containing every derived operator. Thus, if we denote derived operators by terms, the derived signature of `MULT` looks like

```
[ Zero NzNat < Nat ]
op 0 : -> Zero
op s : Nat -> NzNat
op _+_ : Nat Nat -> Nat
op _*_ : Nat Nat -> Nat
...
op s(0) : -> NzNat
...
op s(s(X)) : Nat -> NzNat
...
```

[2] Actually, not so easy. You have to calculate a least sort.

```
op s(0) + X : Nat -> Nat
...
```

which, of course, we cannot write out fully, since this signature is infinite.

Signature Morphisms A signature morphism is a structure-preserving mapping from a signature to another. For example,

```
signature {
  [ S < S' ]
  *[ H ]*
  bop f : H -> S
}
```

may be mapped to

```
signature {
  [ T < T', U ]
  *[ K < K' ]*
  bop g : K -> T
}
```

via the mapping

S	$\rightarrow$	T
S'	$\rightarrow$	T'
H	$\rightarrow$	K
f	$\rightarrow$	g

which preserves sort ordering and visibility, as well as ranks of operators.

To be definite, given signatures Σ and Σ', a signature morphism is a pair of mappings h_s (of sorts) and h_o (of operators) such that

- h_s preserves and reflects visibility, i.e. S is visible (hidden) iff $h_s(S)$ is;
- h_s preserves sort ordering, i.e. you must have $h_s(S) \leq h_s(S')$ if $S \leq S'$;
- h_s reflects sort ordering on hidden sorts, i.e. you must have $S \leq S'$ if $h_s(S) \leq h_s(S')$;
- h_o preserves ranks, i.e. an operator f with rank $S_1 \ldots S_n \rightarrow S$ must be mapped to an operator with rank $h_s(S_1) \ldots h_s(S_n) \rightarrow h_s(S)$;
- h_o maps behaviourals to behaviourals (`bop` to `bop`), and `op` to `op`; and
- h_o is surjective on behaviourals, i.e. for any behavioural operator g in the target signature, there must be a (behavioural) operator f such that $h_o(f) = g$.

Specification Morphism Signature morphisms take care of signatures. Modules contain axioms as well, and specification morphisms are defined to "preserve" axioms. As an example, consider `MONOID` (Section 5.2.3) and `SIMPLE-NAT` (Section 2.2). If `SIMPLE-NAT` indeed specifies natural number addition, it is associative and has identity `0`, hence constitutes a monoidal structure. This fact can be confirmed by

1. Specifying a mapping of sorts and operators that is a signature morphism, and
2. Showing that, under that mapping, the (translated) axioms of `MONOID` hold in `SIMPLE-NAT`.

A specification morphism is a signature morphism that satisfies such conditions as 2. above. A precise definition would require the precise notion of satisfaction, which is beyond this exposition (note that you have to consider transition relations and behavioural equivalence, in addition to equality). In its place and for intuitive illustration, we shall elaborate on the above monoid example, after introducing concrete language constructs.

8.2.3 View

A *view* specifies ways to bind actual parameters to formal parameters. Given a parameter, i.e. a module T, and a module M, a view V from T to M is a specification morphism from T to M', the derived specification of M, with one modification: the operator mapping must be surjective only for non-derived behavioural operators (i.e., those originally in M). To spell out the definitions of the previous section, therefore, a view is a pair of structure-preserving mappings of sorts and operators, under which axioms of T hold in M' — hence in M, since M' does not contain extra axioms.

To instantiate a parameterised module, it is not enough to supply an actual parameter, since there are in general several ways to replace sorts with sorts and operators with operators. You need a view. A view declaration consists of a parameter module, an actual parameter module, and a set of maps that constitute a specification morphism.

Syntax 63: *view*
  `view` *view_name* `from` *module_name* `to` *module_name* "{"
    view_element $*_{,}$
  "}"

view_element	::=	*map*
	|	*variable_declaration*

A view name is a character string. The module names after `from` and `to` are those of the parameter module and the actual parameter respectively. Maps are as follows.

map	::=	`sort` *sort_name* "-> *sort_name*
	|	`hsort` *sort_name* "-> *sort_name*
	|	`op` *operator_reference* "->" *operator_reference*
	|	`bop` *operator_reference* "->" *operator_reference*

A sort name may be qualified by a module name or a parameter name (Section 8.1.3), and has to be visible for `sort`, and hidden for `hsort`. An operator reference is an operator name, or a term that denotes a derived operator. An operator reference may also be qualified. And needless to say, `op` is for non-behavioural operators, and `bop` is for behaviourals.

For a source operator, only operator names, or terms that denotes original operators, is allowed. Using the example of `MULT` with variables `X` and `Y`,

```
_*_
X * Y
```

are allowed, while

```
s(0)
s(X) * Y
X * X
```

are not.

A variable declaration is as in Section 4.1, like

```
vars X Y : Nat
```

where sort names are those in the *source* module.

Warning You may sometimes omit commas between view elements, but it is safer always to put them.

For a view declaration to define a specification morphism, the maps must satisfy the conditions for signature morphisms, and the actual parameter must satisfy the translated axioms of the formal (See 8.2.2). And we remark that

- If operators are overloaded, a single map determines a correspondence between all the overloaded operators at once;
- A view may be endomorphic, and not necessarily be the identity mapping at that; and
- A view need not be onto, nor one-to-one.

Warning The system only checks syntactic conditions. To check theory inclusion, you need a powerful theorem prover, which is lacking in the current system.

Warning Equational theory attributes (Section 7.1) act like axioms when views are considered: associativity, for example, is an axiom that must hold in the actual parameter. In the spirit of checking only syntactic conditions, the system ignores these attributes.

Warning The system does not check every syntactic conditions.

8.2.4 Views Galore

We illustrate the definitions and remarks by several examples. A first example maps `MONOID` to `SIMPLE-NAT` (Sections 5.2.3 and 2.2).

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

This mapping is certainly a signature morphism, since, by replacing `M` with `Nat`, the ranks of `e` and "`_*_`" match `0` and "`_+_`" respectively. And we can show that, for variables `X`, `Y`, and `Z` of sort `Nat`, the equations

```
eq (X + Y) + Z = X + (Y + Z) .
eq 0 + X = X .
eq X + 0 = X .
```

hold under the theory of `SIMPLE-NAT` (but this requires a non-trivial proof; we shall come to this later). Now consider another monoid, natural numbers under multiplication:

```
view NAT-AS-MONOID' from MONOID to MULT {
  sort M -> Nat,
  op e -> s(0),
  op _*_ -> _*_
}
```

`MULT` was in Section 6.3.1. Here `M` is mapped to the same sort `Nat` as before, but the operators are mapped differently. Note first that "`_*_`" refers to two distinct operators: over `M` and `Nat`. Further note that the constant `e` is mapped to `s(0)`, which is *not* a constant declared in `MULT`. It is a derived operator, and this view shows why a view is defined to be a mapping to the derived specification. To show that `NAT-AS-MONOID'` defines a specification morphism, we need to prove

```
eq (X * Y) * Z = X * (Y * Z) .
eq s(0) * X = X .
eq X * s(0) = X .
```

Note, finally, that we may as well replace `SIMPLE-NAT` by `MULT` as the target of `NAT-AS-MONOID`, since `MULT` imports every declaration of `SIMPLE-NAT`. This fact confirms the observation that there can be more than one views between two modules.

We now turn to somewhat artificial examples. For the sake of brevity, let us define a module without axioms:

```
module NAT-INT {
  [ Nat < Int ]
  op _+_ : Nat Nat -> Nat
  op _+_ : Int Int -> Int
}
```

Then we have a view

```
view ID from NAT-INT to NAT-INT {
  sort Nat -> Nat,
  sort Int -> Int,
  op _+_ -> _+_
}
```

which is just the pair of identity mappings. Note that the operator map denotes two maps, for "_+_" on `Nat` and that on `Int`. This is an example of overloaded operators being bundled into one (Section 8.1.4).

We have other endomorphisms.

```
view U from NAT-INT to NAT-INT {
  sort Nat -> Nat,
  sort Int -> Nat,
  op _+_ -> _+_
}

view L from NAT-INT to NAT-INT {
  sort Nat -> Int,
  sort Int -> Int,
  op _+_ -> _+_
}
```

`U` collapses `Nat` and `Int` into `Nat`. Recall that the conditions on signature morphisms only require preservation of weak sort ordering. `Nat < Nat` holds[3], so `U` does define a signature morphism. The case of `L` is similar, collapsing the two sorts into `Int`. Even with relatively simple structures like this, you have a couple of different views between modules.

8.2.5 Not Quite a View

We show why the conditions on views (see Section 8.2.2, on specification morphisms) are reasonable. It is easy to see why you have to preserve ranks and sort ordering. For example, given a module

```
module FUNCTION {
```

[3]Remember that `<` denotes less than *or equal to*.

```
    [ S T ]
    op f : S -> T
  }
```

and a self-mapping

```
  sort S -> T
  sort T -> S
  op f -> f
```

it is hard to know where to locate the image of `f`.

The conditions on hidden sorts and behavioural operators ensure the preservation of visibility — mapping hiddens to hiddens, behaviourals to behaviourals — and observations. The following counterexample illustrates the reason why views must reflect hidden sort ordering and be surjective on behavioural operators. Recall the module `COUNTER` (Section 5.2.3), and consider a module of bank accounts:

```
  module SUBTRACT {
    protecting (SIMPLE-NAT)
    op _-_ : Nat Nat -> Nat
    vars N N' : Nat
    eq 0 - N = 0 .
    eq N - 0 = N .
    eq s(N) - s(N') = N - N' .
  }

  module* ACCOUNT {
    protecting (SUBTRACT)
    *[ Account ]*
    bops deposit withdraw : Account Nat -> Account
    bops balance point : Account -> Nat
    var N : Nat
    var A : Account
    eq balance(deposit(A, N)) = balance(A) + N .
    eq balance(withdraw(A, N)) = balance(A) - N .
    eq point(deposit(A, N)) = point(A) + s(0) .
  }
```

"`_-_`" on `Nat` is the usual pseudo-subtraction (you have already seen this definition in Section 6.2.6, as a preparation to declare `GCD`). An account has a balance and a "bonus point". The bonus point increases each time you make a deposit. If you identify a counter reading with the amount of your fortune, it seems plausible to define a view, by the following mapping.

```
  sort Zero -> Zero
  sort NzNat -> NzNat
  sort Nat -> Nat
  op 0 -> 0
  op s -> s
  op _+_ -> _+_
  var N : Nat
  var C : Counter
  hsort Counter -> Account
  bop add(N, C) -> deposit(C, N)
  bop read -> balance
```

Note first the appearance of `Zero`, `NzNat`, et al. A view defines a mapping of all the sorts and operators, including those declared in imported modules[4]. As a result, views easily become large. And quite often, they contain a large uninformative set of maps — the identitiy maps, like here. There is a way to cut them out. See Section 8.2.6.

Note also the usage of variables. In mapping

```
bop add : Nat Counter -> Counter
```

to

```
bop deposit : Account Nat -> Account
```

you need to change the order of arguments. Variables and derived opeartors are used to state this permutation. In the term `deposit(C,N)`, the variable `C` is interpreted as that of `Account`, according to the sort map from `Counter` to `Account`.

Let us return to conditions on views. This mapping does not constitute a view, since `withdraw` and `point` are not in the image of the mapping. This oversight is harmful. It can be shown that in `COUNTER` the behavioural equation

```
beq add(m + n, c) = add(n, add(m, c)) .
```

holds for any `m`, `n` of sort `Nat` and for any `c` of sort `Counter`, while its counterpart

```
beq deposit(a, m + n) = deposit(deposit(a, m), n) .
```

in `ACCOUNT` does not. The reason is that the observation `point(A:Account)` distinguishes the two terms: thus behavioural equivalence is not preserved under this mapping — even though the (single) axiom of `COUNTER` is preserved.

You should have ensured that no additional observation is imposed on (the image of) `Counter`. The conditions on views are sufficient for this purpose. And note well that, if you pay respect to the conditions, behavioural equivalence is preserved whenever the *explicitly given*, *finite* axioms are preserved.

8.2.6 Succinct Views

Fully to spell out the necessary maps may lead to unwieldy views. In the previous section, you already get a hunch that imported sorts and operators induce explosion. In fact, it is possible to omit some obvious maps: the system guesses the missing pieces by the following rules. Unless explicitly written otherwise,

1. If the source and target modules have common submodules, all the sorts and modules declared 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 namesakes; and

3. If the source module has a single sort and the target has a *principal* sort, the single sort is mapped to the principal.

The rules 1. and 2. are obviously helpful: they allow you to omit the obvious.

A principal sort is declared within the header part of a module declaration, as follows.

[4] Even implicitly imported modules. We ignore them here, for your health.

Syntax 64: *module with principal sort*
`module` *module_name* ["(" *list_of_parameters* ")"] `principal-sort` *sort_name* "{"
module_element ∗
"}"

`principal-sort` may be abbreviated to `p-sort`. A sort name refers to a sort that is declared within the module, or in an imported module. Let us show some examples. If you have

```
module* ONE { [ Elt ] }
```

and

```
module! SIMPLE-NAT' principal-sort Nat {
  ... the same as SIMPLE-NAT
}
```

it is possible to declare a view

```
view V from ONE to SIMPLE-NAT' { }
```

which apparently is an empty mapping. This is equivalent to

```
view V from ONE to SIMPLE-NAT' { sort Elt -> Nat }
```

Since great many views are from single-sort modules, principal sorts are very useful for making views compact.

If so declared, a sort is principal in and only in that module, and principality is *not* obtainable via importation[5].

Warning The current version does not support all the mechanisms explained here. Identifying sorts or operators by name is sometimes tedious — you may have to probe the depth of the module hierarchy. Or controvertial. Consider, e.g., "_+_" for natural number addition and for direct summation of sets.

It is also possible to shorten views by importation.

Warning At the time of writing, it is not possible. We have not even worked out a relevant syntax.

8.3 Binding Parameters

8.3.1 Instantiation of Parameterised Modules

An instance of a parameterised module is created by binding actual parameters to formals. The process of binding is called *instantiation*. The result of instantiation is a new module, obtained by replacing occurrences of parameter sorts and operators by actuals. If, as a result of instantiation, a module is imported twice, it is assumed to be imported once, and shared throughout (Section 8.1.2).

Syntactically, instantiation binds a view to a parameter.

Syntax 65: *instantiating parameterised module*
module_name "(" *binding* $+_{,}$ ")"

There are two distinct kinds of binding constructs.

[5] You may buy out Monaco, but it is another matter.

Instantiation by Declared Views You may bind an already declared view to a parameter.

binding ::= *parameter_name* "<=" *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`, and
2. The common submodules of `M` and `M'` are shared.

Instantiation by Ephemeral Views It is possible to declare and bind a view simultaneously.

binding ::= *parameter_name* "<=" *view*
view ::= `view to` *module_name* "`{`" *view_element* ∗ "`}`"

where "`{`", "`}`" encloses the same constructs as in view declarations. The view is valid only within the instantiation. For example,

```
module NAT-ILIST {
  protecting (ILIST(IDX <= view to SIMPLE-NAT { sort Elt -> Nat },
                    DAT <= view to SIMPLE-NAT { sort Elt -> Nat }))
}
```

binds a `SIMPLE-NAT` to the parameter `IDX` and `DAT`, via an identical view from `ONE` (see Section 2.2 for `SIMPLE-NAT`, Section 8.2.1 for `ILIST` and `ONE`). The code is equivalent to

```
view V from ONE to SIMPLE-NAT { sort Elt -> Nat }

module NAT-ILIST {
  protecting (ILIST(IDX <= V, DAT <= V))
}
```

An ephemeral view may be as succinct as a everlasting counterpart, with resort to principality and titular coincidence.

To make the code more concise, it is possible to identify parameters by positions, not by names. `NAT-ILIST` may then be defined by

```
module NAT-ILIST {
  protecting (ILIST(V,V))
}
```

or, more ephemerally,

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

Note that in the latter code "`view to`" constructs are omitted.

8.3.2 Parameters as Imports

A parameter may be regarded as a submodule. If so regarded, it poses a question of importation modes (Section 2.5) and module contexts (Section 8.1.1). Indeed, it is possible to specify an importation mode for a parameter, as in

```
module M (extending X :: T) { ...
```

Available modes are the same as with plain importations: `protecting`, `extending`, and `using`. The default mode is `protecting`. You can check by feeding `ILIST` (Section 8.2.1) and print parameters as follows.

```
CafeOBJ> select ILIST

ILIST(IDX,DAT)> show params
argument IDX : protecting ONE
argument DAT : protecting ONE

ILIST(IDX,DAT)>
```

Note in passing that the new prompt shows parameters as well as the module name `ILIST`.

> **Warning** As stated in Section 2.5, the system does not care if you have chosen an incorrect importation mode. Since various modules are to be bound to parameters, you should be especially cafeful.

As to contexts, a parameter is included in the context graph of the module. The difference from plainly imported modules is that the edge has a name. For example, if `M` imports `M'`, the edge is anonymous

$$\mathtt{M} \longrightarrow \mathtt{M'}$$

while if `M` has a parameter "`X :: M'`", the edge is

$$\mathtt{M} \xrightarrow{\mathtt{X}} \mathtt{M'}$$

Moreover, since a module may have the same parameter with different names, a context graph is in general a hypergraph.

Some aspects of instantiation is better explained in terms of such graphs. Instantiation is then a process of subgraph replacement, and the usual considerations — indentifying which edges with which, determining targets of edges, avoiding dangling edges, and so on — are relevant to the process of instantiation.

8.3.3 Parameter Passing

Since a parameterised module is a module, it may be imported. You may enrich `ILIST` (Section 8.2.1), adding an operator that counts entries.

```
module ILIST-EXT {
  protecting (ILIST)
  protecting (SIMPLE-NAT)
  op #_ : Ilist -> Nat
  var I : Elt.IDX
  var D : Elt.DAT
  var L : Ilist
  eq # put(I,D,L) = s(0) + (# L) .
  eq # empty = 0 .
}
```

The definition of "`#_`" is obvious, but this module is interesting in that it imports a parameterised module without instantiation. Let us see how the system treats such a module. After feeding `ILIST-EXT`, you get

```
CafeOBJ> select ILIST-EXT

ILIST-EXT(IDX, DAT)> show params
argument IDX.ILIST : protecting ONE
argument DAT.ILIST : protecting ONE

ILIST-EXT(IDX, DAT)> show param IDX
module* IDX.ILIST :: ONE {
  imports {
    protecting (BOOL)
  }
  signature {
    [ Elt ]
  }
}

ILIST-EXT(IDX, DAT)>
```

Look first at the new prompt. In the previous section, after the parameterised `ILIST` was `selected`, the prompt changed to

```
ILIST(IDX, DAT)>
```

to indicate that the current module had two parameters. The `selection` here of `ILIST-EXT` also led to the prompt suggesting parameters. As the `show` commands showed, this module indeed has parameters `IDX` and `DAT`, which are both the module `ONE`.

Since `ILIST-EXT` is thus parameterised, you can instantiate it, just as you instantiate `ILIST`. For example,

```
module NAT-ILIST-EXT {
  protecting (ILIST-EXT(IDX <= view to SIMPLE-NAT { sort Elt -> Nat },
                        DAT <= view to SIMPLE-NAT { sort Elt -> Nat }))
}
```

defines a module of enriched indexed lists whose indices and data are both natural numbers.

In importing parameterised modules, you can bind parameters to parameters — "passing parameters", as in the following construction.

```
module ILIST' (X :: ONE) {
  protecting (ILIST(X, X))
}
```

Remember that this shorthand notation means that the default view — i.e., identity mapping —, from `X` (i.e. `ONE`) to `IDX` and `DAT` (both `ONE`) is bound to the parameters `IDX` and `DAT`. The result of the instantiation is the sort of indexed lists whose indices and data belong to the same (parametric) sort. Let us show a session that reveals what is going on.

```
CafeOBJ> module ILIST' (X :: ONE) {
  protecting (ILIST(X, X))
}
-- defining module ILIST'_.*.,,,,,,,_,,*._* done.

CafeOBJ> select ILIST'

ILIST'(X)> show params
```

```
argument X : protecting ONE

ILIST'(X)> show subs
protecting(ILIST(IDX <= X, DAT <= X))

ILIST'(X)>
```

The module ILIST' has a parameter X and imports an instance of ILIST to whose parameters X are bound. We continue the session and create an instance of ILIST'.

```
ILIST'(X)> module ILIST'-NAT {
  pr (ILIST'(X <= view to SIMPLE-NAT { sort Elt -> Nat }))
}

-- defining module ILIST'-NAT,,,,,,,*,,,,,,,*._* done.
ILIST'(X)>
```

ILIST'-NAT is a module of indexed lists whose indices and data are both natural numbers — which is the same module as ILIST-NAT in the previous section.

Finally, you may use this parameter passing mechanism to get an alternative definition of enriched indexed lists, as follows.

```
module ILIST-EXT (IDX :: ONE, DAT :: ONE) {
  protecting (ILIST(IDX, DAT))
  protecting (SIMPLE-NAT)
  op #_ : Ilist -> Nat
  var I : Elt.IDX
  var D : Elt.DAT
  var L : Ilist
  eq # put(I,D,L) = s(0) + (# L) .
  eq # empty = 0 .
}
```

8.3.4 Parameters with Parameters

There are also cases when parameter modules themselves are parameterised. Consider the following example.

```
module* ONE { [ Elt ] }
module* STACK (X :: ONE) {
  [ Stack ]
  op push : Elt Stack -> Stack
}

module TRAY (Y :: STACK) {
  [ Tray ]
  op tray : Stack Stack -> Tray
  ops in-tray out-tray : Tray -> Stack
  op in : Elt Tray -> Tray
  var E : Elt
  vars S S' : Stack
  eq in(E, tray(S,S')) = tray(push(E,S),S') .
  trans tray(push(E,S),S') => tray(S, push(E, S')) .
  eq in-tray(tray(S,S')) = S .
  eq out-tray(tray(S,S')) = S' .
}
```

The module TRAY is intended to describe the desk of an important executive. The parameter STACK of this module itself has a parameter ONE (the same as the one in Section 8.2.1).

There are two ways to instantiate TRAY:

a. Instantiate Y, or

b. Instantiate X of Y.

In case of a., Y should be bound by a module which has the same module ONE as parameter. For example, if you have defined

```
module LIST (X :: ONE) {
  [ NeList < List ]
  op nil : -> List
  op __ : Elt List -> NeList
  op __ : Elt NeList -> NeList
  op head_ : NeList -> Elt
  op tail_ : NeList -> List
  var E : Elt
  var L : List
  eq head (E L) = E .
  eq tail (E L) = L .
}
```

you may bound LIST[6] to the parameter Y, as

```
view LIST-AS-STACK from STACK to LIST {
  sort Stack -> NeList,
  op push -> __
}

module LIST-TRAY {
  protecting (TRAY(Y <= LIST-AS-STACK))
}
```

The result is a module with parameter X (originated from LIST). This module defines trays of lists of Elt. Note that LIST-AS-STACK omits the map from Elt of X to itself. This is an example of implicit submodule mapping (cf. Section 8.2.6). You may further instantiate LIST-TRAY, by binding something to X — which is ONE, the module of an arbitrary set.

In case of b., you bind something to X (of STACK), which is the module ONE. For example,

```
module NAT-TRAY {
  protecting (TRAY(X.Y <= view to SIMPLE-NAT { sort Elt -> Nat }))
}
```

creates a module whose trays consist of stacks of natural numbers — trays of mathematicians, perhaps. Note that X.Y is a qualification of a parameter name by another parameter name (cf. Section 8.1.3). NAT-TRAY is not parametric; as a result of the instantiation, STACK becomes in effect a plain submodule of NAT-TRAY.

8.3.5 Qualifying Parameter Names

Parameter names may conflict, when a parameterised module imports another parameterised module. To disambiguate parameter names, you may qualify them by module names (Section 8.1.3). For

[6] It has been a ritual to embed elements as one-length lists with a subsort declaration. This practice is not recommended since you may unable to get pushouts as desired. See the Haxhausen's paper in AMAST'96 for detail.

example, in

```
module A ( X :: B ) { ... }
module C ( X :: D ) {
  protecting (A)
  ...
}
```

X is both the parameter name of A and that of C, and is ambiguous in C. You may refer to X of A by "X.A". The plain X, in C, refers to X of C. This means that a sort or operator name may be qualified by a parameter name which itself is qualified by a module name. Since a term may be qualified by a sort name also (see Section 3.3.2), a chain of qualifications such as

```
(f(a)):S.P.M
```

is possible. If there is any ambiguity, qualifier "." is considered right associative.

8.4 Module Expression

Syntactically, an instantiation of parameterised modules is an expression involving module names. There are other kinds of such expressions. Expressions over module names are called *module expressions*. A module expression represents a module, and may appear in import declarations or in several commands. Here we summarise the syntax of module expressions.

8.4.1 Module Name

The simplest module expressions are module names.

Syntax 66: *module constant* *module_name*

A module represented by a module name is the module itself.

8.4.2 Renaming

Renaming is to change sort or operator names, creating a new module.

Syntax 67: *renaming* *module_expression* "*" "{" *map* $*_{,}$ "}"

where a map is

map ::= `sort` *sort_name* "->" *sort_name*
| `hsort` *sort_name* "->" *sort_name*
| `op` *operator_name* "->" *operator_name*
| `bop` *operator_name* "->" *operator_name*

Maps are as expected: `sort` maps visible sorts, `hsort` invisibles, `op` ordinary operators, `bop` behaviourals, and `param` parameters. Sort and operator names should be familiar now. Source names may be qualified; target names may not (they are supposed to be brand-new names).

Warning Commas may be omitted.

Note that renaming creates a new, different module, even if it has isomorphic models. A consequence is that a renamed module is not shared (cf. Section 8.1.2).

Warning Renaming is meant to describe a bijective signature morphism (Section 8.2.2). The current implementation allows more flexible renaming, however, and you may collapse two distinct sort/operators into one. If you do give such a renaming, care should be taken.

Renaming is quite often used in combination with instantiations. Since instantiation is a (module-level) operation that generates specific data types from generic ones, it is natural for you to confer names that describe specialised sorts and operators more effectively. For example, in specialising generic indexed lists to those of natural numbers (Section 8.3.1), you may write

```
module NAT-ILIST {
  protecting (ILIST(IDX <= view to SIMPLE-NAT { sort Elt -> Nat },
                    DAT <= view to SIMPLE-NAT { sort Elt -> Nat })
                * { sort IList -> IList-of-Nat })
}
```

In the parameterised `ILIST`, a simple `IList` is okay to denote lists of something. After instantiation, a list is a list of natural numbers, so why not call it as it is.

8.4.3 Module Sum

Combining several modules together is another common module-level operation. The combined whole is called *module sum*.

Syntax 68: *module sum*
module_expression "+" *module_expression* { "+" *module_expression* } *

A module represented by a module sum consists of all the module elements in the summands. If submodules are imported more than once, it is assumed to be imported once, and is shared throughout (Section 8.1.2). This construction is sometimes called *amalgamated sum* or *shared sum*. From this definition, it follows that

```
module A { protecting (B + C) ... }
```

is equivalent to

```
module A { protecting (B) protecting (C) ... }
```

8.4.4 Making Modules

Module expressions are often used in isolation. To create a module, it is often the case that you simply import a module represented by an expression, as

```
module LOVE {
  protecting (...)
}
```

where "`...`" is a module expression. The purpose of this declaration is to give a name to a module defined by the expression. C'est tout. In such a case, an abbreviation is handy.

Syntax 69: `make` *command*
`make` *module_name* "`(`" *module_expression* "`)`"

`make` evaluates the module expression and confer the resulting module with the given name. In fact, this is a shorthand of the protecting importation as above. You may want to try this:

```
CafeOBJ> make LOVE (SIMPLE-NAT)

-- defining module LOVE._* done.
CafeOBJ> show LOVE
module LOVE {
  imports {
    protecting (SIMPLE-NAT)
  }
}

CafeOBJ>
```

Chapter 9

Theorem-Proving Tools

The TRS engine (Chapter 6) may be used as a lightweight theorem prover. In addition, the current version of the system supports a couple of commands that act as theorem-proving tools.

9.1 Open/Closing Modules

There are commands to modify module declarations. To start modifying a module, use the command

Syntax 70: `open` *command* `open` *module_name*

This command enables you to declare new sorts, operators, equations, and so on. Just type declarations after this command. You cannot, however, change or delete the existing declarations.

To finish the modification, use the command

Syntax 71: `close` *command* `close`

9.1.1 Why Opening Modules?

The effects of declarations supplied in between `open` and `close` commands are temporary.

- An `open` command creates a new module. This module contains a copy of declarations in the given module.
- Until the module is `close`d, all the declared sorts, equations, etc. are added to the new module.
- `close` expunges the new module.

So `close` makes all your efforts banish. What is the sense of it all? Well, this open/closing mechanism is an ecological tool to (1) make hypotheses, (2) prove, by evaluation, your favourite theorems upon the hypotheses, (3) finish the proof, and delete the hypotheses, (1') start again with new hypotheses, ...

For example, consider the familiar module `SIMPLE-NAT`, reprinted here.

```
module SIMPLE-NAT {
  [ Zero NzNat < Nat ]
  op 0  : -> Zero
  op s : Nat -> NzNat
```

```
    op _+_ : Nat Nat -> Nat
    vars N N' : Nat
    eq 0 + N = N .
    eq s(N) + N' = s(N + N') .
  }
```

Suppose you want to show that `0` is a right identity of "`_+_`" also. Using the standard structural induction, you can prove it easily, by showing (1) "`0 + 0`" equals `0`, and (2) for any `M`, if "`M + 0`" equals `M`,
"`s(M) + 0`" equals `s(M)`. A score of this proof may be written, using **open** command, as

```
open SIMPLE-NAT
op a : -> Nat .
eq a + 0 = a .
reduce 0 + 0 .
reduce s(a) + 0 .
close
```

When a module is **open**ed, the system changes the prompt beginning with `%`, as

```
CafeOBJ> open SIMPLE-NAT

-- opening module SIMPLE-NAT.. done.

%SIMPLE-NAT>
```

After **open**ing, module elements and various commands can be input. The above score adds a new constant `a` (to represent "any" natural number) and an equation (for induction hypothesis), and invlokes reduction commands (base case and induction step). The system evaluates "`0 + 0`" and "`s(a) + 0`" as usual, and returns the results `0` and `s(a)`.

```
%SIMPLE-NAT> op a : -> Nat .

%SIMPLE-NAT> eq a + 0 = a .

%SIMPLE-NAT> reduce 0 + 0 .
*
-- reduce in % : 0 + 0
0 : Zero
(0.000 sec for parse, 1 rewrites(0.000 sec), 2 matches)
%SIMPLE-NAT> reduce s(a) + 0 .

-- reduce in % : s(a) + 0
s(a) : NzNat
(0.000 sec for parse, 2 rewrites(0.000 sec), 4 matches)
%SIMPLE-NAT> close

CafeOBJ>
```

You may have noticed quizzical "`*`" in the above session. They are not print smudges. When a module is opened, the system processes the subsequent declarations and commands as if they were entered during a normal module declaration, and tries to tell you that it is working furiously (cf. an aside in Section 2.4).

Warning As you may have noticed in the above example, the current implementation requires a blank-period terminator for each declaration (note " ." after the operator declaration above).

The opened module and the current module need not be the same. For example, if the current module is NERD, you can have a session like

```
 NERD> open SIMPLE-NAT

-- opening module SIMPLE-NAT_.* done.

 %SIMPLE-NAT> select BOOL
 ...
 BOOL> close

 NERD>
```

By opening SIMPLE-NAT, the temporary new module is set to be current, as indicated by the prompt. select command switches the current module to BOOL. close restores the previous current module (remember that the effects of declarations and commands during opening are temporary). A convenient way to set the current module while it is opened is selecting the module "%".

```
%SIMPLE-NAT> select %

%SIMPLE-NAT>
```

Warning As you have guessed, the module name "%" refers to the temporary module. You should not use "%" as a module name.

9.1.2 Constant On the Fly

Let us think awhile of the example of the previous section. In the proof score, a constant a was declared by

```
op a : -> Nat .
```

to represent *any* element of sort Nat. Such declarations appear quite often when you are using the evaluation mechanism to prove theorems. To prove that 0 is a right identity, a appeared to calculate the induction step

```
reduce s(a) + 0 .
```

To make proofs more concise, you may use on-the-fly declarations of constants, similar to variable declarations on the fly (Section 4.1).

Syntax 72: *constant declaration on the fly*
constant_name " : " *sort_name*

where a constant name is a character string that starts with the backquote "‘". The effects of declarations are temporary, like the case of variables. Using this construct, it is possible to invoke reduction commands, without opening modules, as

```
CafeOBJ> select SIMPLE-NAT

SIMPLE-NAT> reduce 0 + 0 .

-- reduce in SIMPLE-NAT : 0 + 0
0 : Zero
```

```
(0.000 sec for parse, 1 rewrites(0.000 sec), 1 matches)
SIMPLE-NAT> reduce s('a:Nat) + 0 .

-- reduce in SIMPLE-NAT : s('a:Nat) + 0
s('a:Nat + 0) : NzNat
(0.000 sec for parse, 1 rewrites(0.000 sec), 6 matches)
SIMPLE-NAT>
```

This time, the hypothesis is implicit: instead of stating and using it explicitly, you take note of it in analysing the result of the second evaluation.

Warning In fact, you get quite similar effects by using variables on the fly, instead of constants. Although evaluation is a procedure for ground terms, the system allows variables within terms to be reduced. Here is an example corresponding to the second reduction above.

```
SIMPLE-NAT> reduce s(X:Nat) + 0 .

-- reduce in SIMPLE-NAT : s(X:Nat) + 0
s(X:Nat + 0) : NzNat
(0.000 sec for parse, 1 rewrites(0.000 sec), 6 matches)
SIMPLE-NAT>
```

But note that the system does not treat non-ground terms as such. It treats variables as if they were constants, and would not attempt unification.

9.2 Applying Rewrite Rules

In Section 9.1, it was shown that reductions can be used to prove some theorems. For more demanding theorems, they are inadequate. Consider the additive group theory written in CafeOBJ.

```
module* GROUP {
  [ G ]
  op 0 : -> G
  op _+_ : G G -> G { assoc }
  op -_ : G -> G
  var X : G
  eq 0 + X = X .
  eq (- X) + X = 0 .
}
```

Suppose you want to show that "`- X`" is also a right inverse (so *the* inverse). Using `open`, you may write a score as

```
open GROUP
op a : -> G .
reduce a + (- a) .
```

or, using a constant on the fly, just

```
reduce in GROUP : 'a:G + (- 'a) .
```

But the result of reduction is simply

```
CafeOBJ> reduce in GROUP : 'a:G + (- 'a) .
```

```
-- reduce in GROUP : 'a:G + - 'a:G
'a:G + - 'a:G : G
(0.000 sec for parse, 0 rewrites(0.000 sec), 8 matches)
CafeOBJ>
```

which is no wonder, since no equation has a matching lefthand side. In fact, a standard proof goes as follows.

```
a + (- a) =
0 + a + (- a) =
(- (- a)) + (- a) + a + (- a) =
(- (- a)) + 0 + (- a) =
(- (- a)) + (- a) =
0
```

where we omitted parentheses since "_+_" was declared associative. This proof requires two "reverse" applications of rewrite rules.

9.2.1 Start, Then Apply

To accommodate such proofs as above, CafeOBJ supports the following set of commands.

> **Syntax 73: `start`** *command*
> `start` *term* "`.`"

where a term is a term of the opened module (Section 9.1), or in the current context (Section 8.1.1). This command sets the focus on the given term, which may be printed and referred to in the subsequent commands. `show` command (Section 1.3.3) with `term` as argument prints the term just selected.

> **Syntax 74: `apply`** *command*
> `apply` *action* [*substitution*] *range selection*

where the action, substitution, range and selection are as follows.

action	::=	`reduce`
	—	`exec`
	—	`print`
	—	*rewrite_rule*

where a rewrite rule is

> **Syntax 75:** *rule specification in* `apply` *command*
> ["+" | "-"][*module_name*] "`.`"{*number*|*label*}

If the action is `reduce` or `exec`, the (sub)term specified by the range and selection is reduced. If it is `print`, the (sub)term is printed. If it is a rewrite rule, the chosen rule is applied, once, to the subterm. In the last case, the numbered or labelled rule of the module (if omitted, of the opened module) is used from left to right (if the sign is "+" or none), or right to left (if "-"). We will first show a couple of examples in Section 9.2.2 and say more on this construct later (Section 9.2.4).

Substitutions are of the form

> **Syntax 76:** *substitution specification in* `apply` *command*
> `with` { *variable* "=" *term* } +,

This option is necessary only when the action is a rewrite rule, and binds variables that appear in the rule (see the next section for why this option exists).

Ranges and selections are as follows.

range ::= `within` | `at`

`within` means at or inside the (sub)term specified by the selection, while `at` means exactly at the (sub)term.

selection ::= *selector* { `of` *selector* } *

selector ::= `top`
— `term`
— `subterm`
— "`(`" *number_list* "`)`"
— "`[`" *sublist* "`]`"
— "`{`" *subset* "`}`"

`top` and `term` mean the entire term. `subterm` means the pre-chosen subterm (see the next section). For the other alternatives,

- The list of numbers, separated by blanks, within "`(`", "`)`" indicates a subterm by tree search. For example, "`(2 1)`" means the first argument of the second argument; if "`a + (b * c)`" is the term of concern, it means `b`.
- The `[`, `]` selector makes sense only with associative operators, and indicates a subterm in a flattened structure. For example, if "`_*_`" was declared associative and given a term "`a * b * c * d * e`", "`[2 .. 4]`" means "`b * c * d`", while "`[2 .. 2]`" means `b`. The latter case can be abbreviated to `[2]`.
- The `{`, `}` selector makes sense only with associative and commutative operators, and indicates a subterm in a multiset structure. For example, if "`_*_`" was declared associative and commutative, and given a term "`b * c * d * e * f`", both "`{2, 4}`" and "`{4, 2}`" means a subterm "`c * e`", while `{4}` means `e`.

Note that the above selections are indeed subterms. If "`_*_`" is associative, the connectivity is immaterial, in whatever way the term was originally given. If it is commutative in addition, even the order is immaterial.

Warning `top` is obsolete, and is here only for historical curiosity.

9.2.2 Applying Apply Command

`apply` command has a very complicated syntax. It is better to explain its behaviours by examples. We use the group theory presented at the start of this section.

To show "`- X`" is a right inverse, you may use the following sequence of commands.

```
open GROUP
op a : -> G .
start a + (- a) .
apply -.1 at (1) .
apply -.2 with X = - a at [1] .
apply reduce at term .
```

(The numbers of rewrite rules are printed by `show` command. See Section 9.2.4.) After two applications of reversed rules, reduction yields the desired result, as follows.

```
NAT> open GROUP

-- opening module GROUP... done.

%GROUP> op a : -> G
```

```
%GROUP> start a + (- a) .

%GROUP> apply -.1 at (1) .
*result 0 + a + - a : G

%GROUP> apply -.2 with X = - a at [1] .
result - (- a) + - a + a + - a : G

%GROUP> apply reduce at term .
result 0 : G

%GROUP>
```

Look in more detail what happened at the second `apply` command. This command applied

```
eq (- X) + X = 0 .
```

to the subterm `0`. Without saying anything, a variable `X` would be introduced. This situation often occurs when a rule is reversely applied. The substitution "`X = - a`" avoids that.

You can continue, and prove another theorem that `0` is a right identity (so *the* identity). Since we already showed "`a + (- a)`" equals `0`, it is sound to add the equation. That leads to the command sequence

```
eq a + (- a) = 0 .
start a + 0 .
apply -.2 with X = a at (2) .
apply reduce at term .
```

which mirrors the reasoning

```
a + 0 = a + (- a) + a = 0 + a = a
```

9.2.3 Choosing Subterms

If a term has a complicated structure, it is error-prone to select a subterm at once. To make safer selections, a command is supported.

Syntax 77: `choose` *command*
`choose` *selection*

where a selection is as in `apply` (Section 9.2.1). This command sets the focus to the chosen subterm, and its successive application narrows the focus gradually. `show` command with `subterm` argument may be used to print the current subterm in focus.

For example, if you define a module of rings as

```
module* RING {
  [ R ]
  op 0 : -> R
  op _+_ : R R -> R { assoc comm prec: 35 }
  op -_ : R -> R
  op _*_ : R R -> R { assoc prec: 31 }
  vars X Y Z : R
  eq 0 + X = X .
  eq (- X) + X = 0 .
  eq X * (Y + Z) = X * Y + X * Z .
```

```
    eq (Y + Z) * X = Y * X + Z * X .
  }
```

and after opening RING, you get

```
%RING> op f_ : R -> R .

%RING> start ((f f 0) * ((f f f 0) + (- (f f f f 0)))) *
    ((- (f f f f 0)) + ((f f f 0) * ((f f f f 0) *
                                ((f f 0) + ((f f f f f 0) * (f 0)))))) .

%RING> choose (2) .

%RING> show subterm
((- (f (f (f (f 0))))) + ((f (f (f 0))) * ((f (f (f (f 0)))) * ((f (f 0)) +
    ((f (f (f (f (f 0))))) * (f 0)))))) : R

%RING> choose (2) .

%RING> show subterm
((f (f (f 0))) * ((f (f (f (f 0)))) * ((f (f 0)) + ((f (f (f (f (f
    0))))) * (f 0))))) : R

%RING> choose [1 .. 2] .

%RING> show subterm
((f (f (f 0))) * (f (f (f (f 0))))) : R

%RING> show term
(((f (f 0)) * ((f (f (f 0))) + (- (f (f (f (f 0))))))) * ((- (f (f
    (f (f 0))))) + (((f (f (f 0))) * (f (f (f (f 0))))) * ((f (f
    0)) + ((f (f (f (f (f 0))))) * (f 0)))))) : R

%RING>
```

The last command was used to print the entire term that had been started. This command was also useful in the stepper (cf. Section 6.3.2). You may also print the subterm in a tree form, as

```
%RING> show subterm tree
((f (f (f 0))) * (f (f (f (f 0))))) : R
      _*_
     /   \
    f_  f_
     |   |
    f_  f_
     |   |
    f_  f_
     |   |
     0  f_
         |
         0


%RING>
```

The current term, subterm etc. can also be printed by show command, as (cf. Sections 8.1.1, 9.2.5)

```
%RING> show context
-- current context :
[module] %
[special bindings]
  $$term    = (((f (f 0)) * ((f (f (f 0))) + (- (f (f (f (f 0)))))))
      * ((- (f (f (f (f 0))))) + (((f (f (f 0))) * (f (f (f (f 0)))))
      * ((f (f 0)) + ((f (f (f (f (f 0))))) * (f 0)))))) : R
  $$subterm = ((f (f (f 0))) * (f (f (f (f 0))))) : R
[bindings] empty.
[selections]
  1| [ 1 .. 2 ] .
      2| ( 2 ) .
          3| ( 2 ) .
[pending actions]  none.
[stop pattern] not specified.

%RING>
```

9.2.4 Identifying Rewrite Rules

We have not shown how to obtain the number that designates a specific rewrite rule. Returning to GROUP, you may print rewrite rules — or axioms so regarded — as follows.

```
CafeOBJ> open GROUP

-- opening module GROUP_.* done.

%GROUP> show rules
 -- rewrite rules in module : %GROUP
  1 : eq 0 + X = X
  2 : eq - X + X = 0

%GROUP>
```

Rules of any module may be printed, by supplying an extra argument.

```
%GROUP> show rules RING
 -- rewrite rules in module : RING
  1 : eq 0 + X = X
  2 : eq - X + X = 0
  3 : eq X * (Y + Z) = X * Y + X * Z
  4 : eq (Y + Z) * X = Y * X + Z * X

%GROUP>
```

It is also possible to print all the rules, including those declared in imported modules, by specifying all, as in

```
%GROUP> show all rules
```

The numbering of the above is used to specify rewrite rules. For example, you encountered in Section 9.2.2 a command

```
%GROUP> apply -.1 at (1) .
```

The specified rule, according to the output above, was

```
  1 : eq 0 + X = X
```

and the minus sign was added to apply this rule in reverse. You may decorate a rule number with a module name, like

```
%GROUP> apply -GROUP.1 at (1) .
```

which will apply the same rule as before. Axiom labels may also be used to designate rules. For example, if the above equation were labelled as

```
eq [l-id] : 0 + X = X .
```

the `apply` command might as well be

```
%GROUP> apply -.l-id at (1) .
```

Warning Do not put blanks anywhere in between the sign, module name, period, number or label. Memorise "a rule spec hates vacuum".

Finally, `show` command with `rule` (not `rules`) may be used to print a single rule. This command is useful when the number of rewrite rules is large. An example:

```
%GROUP> show rule .2
rule 2 of the last module
  - X + X = 0
(This rule rewrites up.)
```

Do not forget the period before 2. Arguments to this command are rule specifications, and "`.2`" is of rule specification form, according to the definition in Section 9.2.1.

"`show rule`" and "`show rules`" may be invoked when no module is opened. In such a case, the rule(s) of the current module are printed.

9.2.5 Appyling Conditionals

When the action is a rewrite rule, `apply` command tries to apply just one rule and immediately returns the control to you. What if the rule is conditional? Recall `GCD` (Section 6.2.6):

```
CafeOBJ> open GCD

-- opening module GCD.. done.

%GCD> start gcd(s(0),s(s(0))) .

%GCD> show rules
 -- rewrite rules in module : %GCD
  1 : ceq gcd(N,M) = gcd(M,N) if N < M
  2 : eq gcd(N,0) = N
  3 : ceq gcd(s(N),s(M)) = gcd(s(N) - s(M),s(M)) if not N < M

%GCD> apply +.1 at term .
shifting focus to condition
-- condition(1) s(0) < s(s(0)) : Bool

%GCD> show context
```

```
-- current context :
[module] %
[special bindings]
  $$term    = (s(0) < s(s(0))) : Bool
  $$subterm =  $$term
[bindings] empty.
[selections]  empty.
[pending actions]
  1| in gcd(s(0),s(s(0)))  at top
   | rule ceq gcd(N,M) = gcd(M,N) if N < M
   | condition s(0) < s(s(0))  replacement gcd(s(s(0)),s(0))
[stop pattern] not specified.

%GCD>
```

It looked certain that the rule 1 would apply, so you tried it. The system then told that, since that rule was conditional, it had to resolve the condition first.

show command with argument were partly explained in Sections 8.1.1 and 9.2.3. What was remained was the misterious heading "[pending actions]". This part is used exactly in a situation like now, and contains terms to be further reduced after a condition — the current focus — has been evaluated.

The instance of the condition requires rules in SIMPLE-NAT+ (Section 6.2.6), so you have to continue as follows.

```
%GCD> show rules SIMPLE-NAT+
 -- rewrite rules in module : SIMPLE-NAT+
  1 : eq 0 - M = 0
  2 : eq N - 0 = N
  3 : eq s(N) - s(M) = N - M
  4 : eq 0 < s(N) = true
  5 : eq N < 0 = false
  6 : eq s(N) < s(M) = N < M

%GCD> apply +SIMPLE-NAT+.6 at term .
-- condition(1) 0 < s(0) : Bool

%GCD> apply +SIMPLE-NAT+.4 at term .
-- condition(1) true : Bool
-- condition is satisfied, applying rule
-- shifiting focus back to previous context
result gcd(s(s(0)),s(0)) : Nat

%GCD>
```

At last the condition was satisfied, and the system rewrote the original term. You may continue, this time applying the rule 3.

```
%GCD> apply +.3 at term .
shifting focus to condition
-- condition(1) not s(0) < 0 : Bool

%GCD>
```

But reduction of "_<_" is obvious so you apply reduce at once.

```
%GCD> apply reduce at term .
```

```
-- condition(1) true : Bool
-- condition is satisfied, applying rule
-- shifiting focus back to previous context
result gcd(s(s(0)) - s(0),s(0)) : Nat

%GCD>
```

The original term was further rewritten, and you can continue as before.

As shown by this frustrating example, what `apply` basically offers is a painful step-by-step reasoing. There are ways to skip certain parts at one go. One way is to invoke the action `reduce`, as above. In the case of conditional rewrite rules, there is another, systematic way to do it. Let us redo the above session using this mechanism.

```
%GCD> start gcd(s(0), s(s(0))) .

%GCD> set reduce conditions on

%GCD> apply +.1 at term .
result gcd(s(s(0)),s(0)) : Nat

%GCD> apply +.3 at term .
result gcd(s(s(0)) - s(0),s(0)) : Nat

%GCD> apply +.1 at term .

[Warning]: rule not applied
result gcd(s(s(0)) - s(0),s(0)) : Nat

%GCD>
```

The switch "`reduce conditions`", when on, makes the system to evaluate conditions automatically.

9.3 Matching Terms

Matching is a basic procedure in evaluation (Section 6.1.1). It may be useful by itself as a theorem-proving helper, since

- When you use `reduce` command, it is a good practice to write down the expected result beforehand. If the actual result differs, you immediately notice something is going wrong. A standard way to realise this practice is to use a printable comment "`-->`" or "`**>`" (Section 2.4), as

  ```
  reduce s(s(0)) + s(0) .
  **> should be: s(s(s(0)))
  ```

  The system then prints

  ```
  -- reduce in SIMPLE-NAT : s(s(0)) + s(0)
  s(s(s(0))) : NzNat
  (0.000 sec for parse, 3 rewrites(0.000 sec), 5 matches)
  **> should be: s(s(s(0)))
  ```

  so that you can compare the actual and expected results easily. A more sophisticated mechanism is to invoke a matching procedure. And

- In using `reduction` and `apply` commands for theorem proving, it is necessary to know what rewrite rules are applicable to what terms. You can inspect the rules by `show` command, but if the set of rules is large, it is tedious. A tool to list applicable rules is helpful. "Applicable" here means "having matchable lefthand side".

9.3.1 Match Command

Syntax 78: `match` *command* `match` *term_specifier* `to` *pattern* "`.`"

The last period (preceded by a blank) is necessary. Term specifiers and patterns are as follows.

term_specifier	::=	`top`
	|	`term`
	|	`it`
	|	`subterm`
	|	*term*

where a term is a usual term. `top` or the literal `term` means the term set by `start` command; `subterm` means the term selected by `choose` commands; `it` is the same as `subterm` if `choose` was used, and is the same as `top` otherwise.

Warning Unlike former versions, the current version do not require enclosing parentheses for a term. If you are cautious, put them anyway. They are harmless.

pattern	::=	[`all`] { `rules` | `+rules` | `-rules` }
	|	*term*

where a term is a usual term. `rules`, "`+rules`" or "`-rules`" lists rewrite rules applicable to the specified term. "`+rules`" means the rules with matching lefthand sides, "`-rules`" means those with matching righthand sides, and `rules` means both. If optional `all` is given, all the rules in the current context, including those declared in built-in modules, are inspected.

If a pattern is a term, matching is attempted, and if successful, the matching subsitution is printed. Is a pattern is `rules`, "`+rules`" or "`-rules`", the list of applicable rules, together with corresponding matching substutitions, are printed.

9.3.2 Matching Terms to Terms

We use `GROUP` (Section 9.2) for examples.

```
CafeOBJ> select GROUP

GROUP> match 0 + 0 to X:G + 0 .
-- match success.
 substitution : { X |-> 0 }
-- No more match

GROUP> match 0 + 0 to X:G + Y:G .
-- match success.
 substitution : { X |-> 0, Y:G |-> 0 }
-- No more match

GROUP> match 0 to 0 + 0 .
-- no match

GROUP>
```

As shown above, the system prints the substitution when matching succeeds.

The system recognises equational theory attributes, such as associativity (Section 7.1), and tries all the possibilities, as

```
GROUP> match 0 + 0 + 0 to X:G + Y:G .
-- match success.
 substitution : { X |-> 0, Y:G |-> 0 + 0 }
>> More? [y/n] : y
-- match success :
 * substitution : { X |-> 0 + 0, Y:G |-> 0 }

-- No more match

GROUP>
```

Remember that "_+_" was declared associative. Modulo associativity, there are two ways of matching the above two terms.

Warning It is possible for a term to contain variables, both as a term specifier and as a pattern. For example, it is legal to try

```
match X:G + 0 to 0 + X':G .
```

But this does not acts as expected:

```
GROUP> match X:G + 0 to 0 + X':G .
-- no match

GROUP>
```

`match` is a matching procedure, not a unification. If a term (to the left of "`to`") contains a variable, it is treated as a pseudo-constant. In the above example, `X` acts as a constant, and cannot be substituted by `0`.

9.3.3 Matching Terms to Pattern

We next deal with the cases when the pattern designates a set of rules. For illustration, we add several equations before invoking matching.

```
GROUP> open GROUP

-- opening module GROUP_.* done.

%GROUP> ops a b : -> G .

%GROUP> eq X:G + Y:G = Y + X .

%GROUP> eq a + X:G = a + X + a .

%GROUP> start 0 + a + a .

%GROUP> match term to +rules .
== matching rules to term : (0 + (a + a))
+%.1 is eq 0 + X = X
 substitution : { X |-> a + a }
```

```
+%.3 is eq X + Y:G = Y:G + X
 substitution : { Y:G |-> a + a, X |-> 0 }
+%.4 is eq a + X = a + X + a
 substitution : { X |-> a }

%GROUP>
```

The system lists applicable rules to the term "`0 + a + a`". In this output, a rule is numbered, adorned with the module name "`%`" (the opened module: cf. Section 9.1.1), and a sign. Remember that this form appears in arguments to `apply` command (Section 9.2.4). You may have noticed two differences from the cases of matching terms to terms.

- Not all possibilities are considered. In the above output, only one matching possibility is shown for the rule 3, even though there are two possibilities (see the previous section).
- All the subterms of the given term is under consideration. The rule 4 does not match the whole term, but does match a subterm.

The case where "`-rules`" is specified is similar.

```
%GROUP> match term to -rules .
== matching rules to term : (0 + (a + a))
-%.1 is eq 0 + X = X
 substitution : { X |-> 0 + a + a }
-%.3 is eq X + Y:G = Y:G + X
 substitution : { Y:G |-> 0, X |-> a + a }
-%.20 is eq if true then CXU else CYU fi = CXU
 substitution : { CXU |-> 0 + a + a }
-%.21 is eq if false then CXU else CYU fi = CYU
 substitution : { CYU |-> 0 + a + a }

%GROUP>
```

Warning The last two printed rules are of `BOOL` (actually, a submodule thereof). This rule is always applicable in reverse, since the righthand side is a single variable.

Chapter 10

Proving Module Properties

10.1 Check Command

Some of the module properties are checkable syntactically, within reasonable computation cost.

> **Syntax** 79: `check` *command*
> `check` { `regularity` | `compatibility` } [*module_expression*]
> `check` { `laziness` } [*operator_name*]

A module expression may be omitted if the current module (Section 8.1.1) is set. `regularity` may be abbreviated to `reg` or `regular`, `compatibility` to `compat`, and `laziness` to `lazy`.

Regularity If the first argument is `regularity`, the system checks if the module is regular. Using `INTB'` in Section 3.3.1, you can obtain the following message.

```
CafeOBJ> check regularity INTB'

>> start regularity check ...
>> The following sorts may be required for regularity:
  [ Nat^NonPos < NonPos Nat ]
>> The following operators may be required for regularity:
  _+_ : Nat^NonPos Nat^NonPos -> Nat^NonPos
CafeOBJ>
```

while with `SIMPLE-NAT`, the system reports nothing, as

```
CafeOBJ> check regularity SIMPLE-NAT

>> start regularity check ...
-- signature of module SIMPLE-NAT is regular.
CafeOBJ>
```

Unlike `regularize` command (Section 3.3.1), this command does not modify the signature.

Compatibility If the first argument is `compatibility`, the system checks if the TRS defined by the module is compatible, i.e., every application of every rewrite rule to every well-formed term results in a well-formed term. Note that, in order-sorted rewriting, an application of a rewrite rule may introduce an ill-formed term. For (a contrived) example,

```
module NONCOMPAT {
```

```
  [ S < S' ]
  op a : -> S
  op b : -> S'
  op f : S -> S
  eq a = b .
}
```

is not compatible, since `f(a)`, which is well-formed, rewrites to `f(b)`, which is ill-formed. As this example shows, non-compatibility is caused by a rewrite rule that "raise"s the sort of a term. This is not a necessary condition of compatibility, however. In the absence of the operator `f`, the above module *is* compatible. The system checks a necessary and sufficient condition of compatibility, as

```
CafeOBJ> check compatibility NONCOMPAT

>> started compatibility check:

>> module (corresponding TRS) is NOT compatible:
- rewrite rule
    a = b
  violates the compatibility,
  and following operator(s) can possibly be affected:
    f : S -> S
CafeOBJ> module COMPAT {
    [ S < S' ]
    op a : -> S
    op b : -> S'
    eq a = b .
}

-- defining module COMPAT..._.* done.
CafeOBJ> check compatibility COMPAT

>> started compatibility check:

>> module is compatible.
CafeOBJ>
```

Warning As shown in Section 6.6, the current implementation regards a term like `f(b)` above as *well-formed*, although of a questionable sort. The above definition should be rephrased, in this setting, in terms of error sorts.

Laziness If the first argument is `laziness`, the system checks if the given operator can be evaluated lazily. The operator must be in the current module (which must be set beforehand). If an operator name is omitted, all the operators directly declared in the current module is checked.

Laziness is related to evaluation strategies. The system checks a sufficient condition under which an operator may be evaluated lazily, as explained Section 7.4. Here is an example.

```
CafeOBJ> module LAZY {
  [ S ]
  op f : S -> S
  op g : S -> S
  eq f(f(X:S)) = g(X) .
  eq g(X:S) = X .
}
```

```
-- defining module LAZY..._..* done.
CafeOBJ> select LAZY

LAZY> check laziness

------------------------------------------------------------
              * laziness of operator: (f) *
------------------------------------------------------------
f : S -> S
  * strict on the arguments :  1
  - rewrite strategy: 1 0
  - axioms:
    eq f(f(X:S)) = g(X:S)

------------------------------------------------------------
              * laziness of operator: (g) *
------------------------------------------------------------
g : S -> S
  * may delay the evaluation on the arguments :  1
  - rewrite strategy: 0 1 0
  - axioms:
    eq g(X:S) = X:S

LAZY>
```

Warning The system only checks an apparent sufficient condition, and do not use a detailed strictness analysis as used in functional language compilers. The word "laziness" was chosen humbly and carefully.

10.2 Theorem Proving Techniques

We have already shown a couple of proofs based on the TRS engine of CafeOBJ (Sections 9.1.1, 9.2.2). There a proof score was a sequence of commands, including `open`, `reduce` and `apply`. The system reads the commands, rewrites terms as commanded, and prints the result. Behind such rewrite sequences is equational reasoning: schematically, the reasoning is of the form

$$t_0 = t_1 = \ldots = t_n$$

and the theorem to prove is the equation $t_0 = t_n$.

Another way to prove theorems by evaluation is to state the theorems as terms of sort `Bool`, and to evaluate them. If the terms are reduced to `true`, the theorems are proved. For example, to show

```
eq a + (- a) = 0 .
```

we may try to reduce the term

```
a + (- a) == 0
```

to `true`. This approach does not confer you with much extra power. As explained in Section 4.4.1, in evaluating "_==_", CafeOBJ simply reduces both arguments, and check if they are equal. Apart from the equality checking part, which involves equational theory (Section 7.1), what can be done with "_==_" can be done without. In terms of presentation, however, this approach produces something like

```
...
```

```
%GROUP> start a + (- a) == 0 .
...
%GROUP> apply reduce at term .
result true : Bool
```

which is more appealing to intuition than

```
...
%GROUP> start a + (- a) .
...
%GROUP> apply reduce at term .
result 0 : G
```

where you must mentally compare the actual result with the expected. Especially, when both sides of the equation need evaluation, this approach provides you with a much more compact proof.

10.2.1 Structural Induction

In Section 9.1.1, we have presented a proof score to show that `0` is a right identity of "_+_" under the definition of `SIMPLE-NAT`. The score relies on structural induction.

Strictly speaking, proofs by structural induction only demonstrate that certain properties (called *inductive properties*) hold for the term-generated models of a module, including its initial model. Recall that "`module!`" denotes such a model, while `module` has both tight and loose denotations. Since `SIMPLE-NAT` was declared with `module`, what was proven in Section 9.1.1 is a property that does not necessarily hold when `SIMPLE-NAT` denotes loose models. With this proviso, structural induction is certainly a useful technique, and you are often powerless without it.

Let us show a couple of examples. For the sake of variety, we use the predicate "_==_" here. The following sequence proves associativity of "_+_".

```
** proving associativity of _+_
open SIMPLE-NAT
ops a b c : -> Nat .
reduce 0 + (a + b) == (0 + a) + b .
eq (a + b) + c = a + (b + c) . ** hypothesis
reduce s(a) + (b + c) = (s(a) + b) + c .
close
```

The session goes as follows.

```
CafeOBJ> open SIMPLE-NAT

-- opening module SIMPLE-NAT_.* done.

%SIMPLE-NAT> ops a b c : -> Nat .

%SIMPLE-NAT> reduce 0 + (a + b) == (0 + a) + b .
_*
-- reduce in % : 0 + (a + b) == (0 + a) + b
true : Bool
(0.000 sec for parse, 3 rewrites(0.000 sec), 11 matches)
%SIMPLE-NAT> eq (a + b) + c = a + (b + c) .

%SIMPLE-NAT> reduce s(a) + (b + c) == (s(a) + b) + c .
*
```

```
-- reduce in % : s(a) + (b + c) == (s(a) + b) + c
true : Bool
(0.010 sec for parse, 5 rewrites(0.000 sec), 26 matches)

%SIMPLE-NAT> close

CafeOBJ>
```

Recall that when we gave a view for MONOID (Section 8.2.4), the associativity axiom remained to be proven. This obligation was discharged just now. Similarly, commutativity is proven by

```
** proving commutativity of _+_
open SIMPLE-NAT
ops a b : -> Nat .
reduce 0 + 0 == 0 + 0 .
eq b + 0 = 0 + b . ** (inner) hypothesis
reduce s(b) + 0 == 0 + s(b) .
eq a + X:Nat = X + a . ** (outer) hypothesis -- *
reduce 0 + s(a) == s(a) + 0 .
eq b + s(a) = s(a) + b . ** (inner) hypothesis
reduce s(b) + s(a) == s(a) + s(b) .
close
```

The proof relies on double induction. And in this proof, the "direction" of equations is crucial: in giving the equation marked "*", if you exchanged the sides, the subsequent evaluations would produce `false`. To get `true`, you need reverse applications (by using `apply`).

10.2.2 Nondeterministic Transitions

Since the current implementation of evaluation mechanism is deterministic, it is impossible to prove transition relations by `execute` command alone. In such a case, the transition predicate "_==>_" is often useful. As explained in Section 4.4.2, "_==>_" is supported by "_=(*)=>_", which systematically checks transitions step by step.

Recall CHOICE-NAT (Section 5.2.1), and let us prove a simple fact that

> If t transits to u, $t|t'$ also transits to u, for any t'.

```
CafeOBJ> open CHOICE-NAT

-- opening module CHOICE-NAT.. done.

%CHOICE-NAT> ops a b c : -> Nat .

%CHOICE-NAT> trans a => c .

%CHOICE-NAT> reduce a | b ==> c .

-- reduce in % : a | b ==> c
true : Bool
(0.000 sec for parse, 6 rewrites(0.000 sec), 13 matches)
%CHOICE-NAT> close

CafeOBJ>
```

For another simple example, suppose you want to prove another simple fact that

If t transits to u, $t + t'$ transits to $u + t'$, for any t'.

```
CafeOBJ> open CHOICE-NAT

-- opening module CHOICE-NAT.. done.

%CHOICE-NAT> ops a b c : -> Nat .

%CHOICE-NAT> trans a => c .

_
%CHOICE-NAT> reduce a + b ==> c + b .
*
-- reduce in % : a + b ==> c + b
true : Bool
(0.000 sec for parse, 4 rewrites(0.000 sec), 15 matches)
%CHOICE-NAT> close

CafeOBJ>
```

Note that, when using "_==>_", evaluation should be by `reduce` command, *not* by `execute` cpmmand. Although "_==>_" represents transition relations, the reasoning is within pure equational logic.

There are a couple of switches that control the reduction procedure for "_==>_". One is to get a trace. For illustration, you may rerun the first above session.

```
... declaring a, b, c, and assert a => c as before

%CHOICE-NAT> set exec trace on

%CHOICE-NAT> reduce a | b ==> c .

-- reduce in % : a | b ==> c
** transition step 1-1 *******
(): (a | b)
    =(1)=> a
(): (a | b)
    =(1)=> b
(1): (a | b)
    =(1)=> (c | b)

** transition step 2-1 *******
(): a
    =(2)=> c
** term : c
 matched to the pattern : c
 with the substitution : {}
true : Bool
(0.000 sec for parse, 6 rewrites(0.000 sec), 13 matches)
%CHOICE-NAT>
```

There are three transitions from "`a | b`", that is

```
trans N | N' => N .
trans N | N' => N' .
trans a => c .
```

respectively yielding `a`, `b`, and "`c | b`". The first branch further transits to `c`, which matches the given destination.

10.2.3 Systematic Search for Transition Relations

"_=(*)=>_" can often be used directly. Continuing the example of CHOICE-NAT,

```
CafeOBJ> select CHOICE-NAT

CHOICE-NAT> reduce (0 | s(0)) | (s(s(0)) | (s(s(s(0))) | s(s(s(s(0)))))) =(*)=> s(s(s(0))).

-- reduce in CHOICE-NAT : (0 | s(0)) | (s(s(0)) | (s(s(s(0))) | s(
   s(s(s(0)))))) = ( * ) => s(s(s(0)))
true : Bool
(0.010 sec for parse, 51 rewrites(0.020 sec), 51 matches)
CHOICE-NAT>
```

This was a very easy case, since all the arguments to "_|_" are are irreducible and only pure transition steps are involved. A more demanding case is

```
CHOICE-NAT> reduce (0 | s(0)) + ((s(s(0)) | (s(s(s(0))) + 0)) + s(s(s(s(0)))))
   =(*)=> s(s(0)) + (s(s(s(0))) + s(0)).

-- reduce in CHOICE-NAT : (0 | s(0)) + ((s(s(0)) | (s(s(s(0))) +
    0)) + s(s(s(s(0))))) = ( * ) => s(s(0)) + (s(s(s(0))) + s(0))

true : Bool
(0.000 sec for parse, 35 rewrites(0.000 sec), 78 matches)
CHOICE-NAT>
```

where the the system need to compute "_+_" along the way.

If the search space can be large, "=(*)=>" is a dangerous weapon. In such cases, you may limit the search up to a realistic step, with the set of predicates "=(n)=>" where n is a natural number (in the usual decimal notation) and indicates how many transit steps to take. For example,

```
CHOICE-NAT> reduce (0 | s(0)) | (s(s(0)) | ((s(s(s(0))) + 0) | s(s(s(s(0)))))) =(1)=> s(s(s(0))

-- reduce in CHOICE-NAT : (0 | s(0)) | (s(s(0)) | ((s(s(s(0))) +
     0) | s(s(s(s(0)))))) = ( 1 ) => s(s(s(0)))
[Warning]: =(N)=> terminates, reaching preset maximum count of transitions.
false : Bool
(0.010 sec for parse, 13 rewrites(0.000 sec), 16 matches)
CHOICE-NAT>
```

which is a correct result, since the first argument cannot transit to the second in one step.

Another way to limit the search is by a switch.

```
CafeOBJ> set exec limit 40
```

limits the maximal transition step to 40. The default limit is quite large.

As the above sessions show, "_=(*)=>_" and "_==>_" take into account equations as well as transitions. In most cases and with appropriate evaluation strategies, "_==>_" does give correct answers on transition relations. However, it is in general impossible always to obtain correct answers on transition relations between *equivalence classes* of terms. For example, consider a module

```
module M {
  [ S ]
```

```
    ops a b c d : -> S
    eq a = b .
    trans a => c .
    eq c = d .
  }
```

and a theorem

```
  trans a => d .
```

The system does not evaluate the corresponding term to `true`, as

```
  M> reduce a ==> d .

  -- reduce in M : a ==> d
  > pre normalization : a
      == b
  -- no more possible transition for : a
  b ==> d : Bool
  (0.000 sec for parse, 4 rewrites(0.000 sec), 12 matches)
  M>
```

This is because the system reduces `a` to an irreducible form (`b`) before trying a systematic search. If it had tried a search before normalising, it would have proved the theorem. There is a switch that tells the system to forget about normalisation.

```
  M> set exec normalize on

  M>> reduce a ==> d .

  -- reduce in M : a ==> d

  ** transition step 1-1 *******
  (): a
      =(1)=> c
  < post nomalization : c
      == d
  ** term : d
   matched to the pattern : d
   with the substitution : {}
  true : Bool
  (0.000 sec for parse, 4 rewrites(0.010 sec), 6 matches)

  M>
```

The result was satisfactory. This switch is hopeless if interleaving applications of equations and transitions are necessary. In

```
  module M' {
    [ S ]
    ops a b c d e : -> S
    eq [e1] a = b .
    eq [e2] b = c .
    trans [t] b => d .
    eq [e3] d = e .
  }
```

to prove

```
trans a => e .
```

you have to use `e1`, and only that.

10.2.4 Behavioural Equivalence

Recall `COUNTER` (Section 5.2.3) and let us prove a behavioural equation

```
beq add(N:Nat, add(N':Nat, C:Counter)) = add(N', add(N, C)) .
```

As explained, this equation holds iff two terms act identically on every observation. This theorem may be proved by induction on the length of observations (called *context induction*). Since `add`, `read` are the only "method" and "attribute" respectively, every observation is an application of `read` after a finite number of applications of `add`.

A necessary command sequence is as follows. Since the proof uses associativity and commutativity of "_+_" on `Nat`, which were proven in Section 10.2.1, they are stated as equations.

```
open COUNTER
ops m n p : -> Nat .
op c : -> Counter .
op addn : Counter -> Counter .
eq m + n = n + m .
eq N:Nat + (M:Nat + P:Nat) = (N + M) + P .
reduce read(add(m, add(n, c))) == read(add(n, add(m, c))) .
eq read(addn(add(m, add(n, c)))) = read(addn(add(n, add(m, c)))) .
reduce read(add(p, addn(add(m, add(n, c))))) ==
       read(add(p, addn(add(n, add(m, c))))) .
close
```

Alternatively, you may use "_=*=_" (Section 4.4.3) and get a proof by *coinduction.*

```
CafeOBJ> open COUNTER

-- opening module COUNTER_.*done.

%COUNTER> ops m n : -> Nat .

%COUNTER> op c : -> Counter .

%COUNTER> eq m + n = n + m .
_

%COUNTER> eq N:Nat + (M:Nat + P:Nat) = (N + M) + P .

%COUNTER> reduce add(m, add(n, c)) =*= add(n, add(m, c)) .
*
-- reduce in % : add(m,add(n,c)) =*= add(n,add(m,c))
true : Bool
(0.000 sec for parse, 9 rewrites(0.000 sec), 37 matches)
%COUNTER> close

CafeOBJ>
```

The reduction went as desired, since the system automatically added an equation

```
eq hs1:Counter =*= hs2:Counter = read(hs1:Counter) == read(hs2:Counter) .
```

once "_=*=_" was shown to be a congruence (Section 5.2.4).

As an aside, you may wonder why two axioms are given in these forms: commutativity is stated with constants, while associativity is with variables. The reason is that a straightforward statement of commutativity

```
eq M:Nat + N:Nat = N + M .
```

engenders infinite rewrite sequences. The trick here is to give an axiom that restricts the matching power of the system so that, only in the special instances when you need for this special proof, the axiom is usable as a rewrite rule.

10.2.5 Behavioural Transition

To illustrate proofs on behavioural transitions, we define counters of wobbly numbers.

```
  module W-COUNTER {
    protecting (CHOICE-NAT)
    *[ Counter ]*
    bop read : Counter -> Nat
    bop add : Nat Counter -> Counter
    eq read(add(N:Nat, C:Counter)) = N + read(C) .
}
```

The structure is the same as COUNTER, and the difference is in the imported CHOICE-NAT. We would like to prove a behavioural transition relation

```
btrans add(M:Nat | N:Nat, C:Counter) => add(N, C) .
```

An element of hidden sort behaviourally transits to another iff every observation of the former transits to the same observation of the latter. This definition leads to a proof by context induction, as in the previous section. The session goes as follows.

```
CafeOBJ> open W-COUNTER

-- opening module W-COUNTER_.*done.

%W-COUNTER> ops m n p : -> Nat .

%W-COUNTER> op c : -> Counter .

%W-COUNTER> op addn : Counter -> Counter .

%W-COUNTER> reduce read(add(m | n, c)) ==> read(add(m, c)) .
_*
-- reduce in % : read(add(m | n,c)) ==> read(add(m,c))
true : Bool
(0.010 sec for parse, 6 rewrites(0.010 sec), 32 matches)
%W-COUNTER> trans read(addn(add(m | n, c))) => read(addn(add(m, c))) .

%W-COUNTER> reduce read(add(p, addn(add(m | n, c)))) ==> read(add(p, addn(add(m, c)))) .
```

```
*
-- reduce in % : read(add(p,addn(add(m | n,c)))) ==> read(add(p,addn(
    add(m,c))))
true : Bool
(0.000 sec for parse, 4 rewrites(0.000 sec), 11 matches)
%W-COUNTER> close

CafeOBJ>
```

The command sequence is quite similar to the one in the previous section. Or, relying on "=*=", you may be rest assured by just one reduction

```
%W-COUNTER> reduce read(add(m | n, c)) ==> read(add(m, c)) .
```

which, by itself, is a proof by coinduction. A difference from the case of behavioural equivalence is that the system does not suppy an axiom like

```
eq hs1:Counter =*= hs2:Counter = read(hs1:Counter) == read(hs2:Counter) .
```

so you have to be satisfied by handfeed reduction commands for each "attribute".

Chapter 11

Built-in Modules

You may define any data types in CafeOBJ, so in principle the need for built-in data types does not arise. So is the principle, but considerations about convenience and tolerable performance dictate otherwise. Some frequently used data types are defined in *built-in modules*. In addition, the system relies on a couple of *kernel modules*, which are located near its jugular. A few commands, such `reset` (Section 1.1) are closely related to these modules.

Listed below are the kernels and built-ins. (As explained in Section 1.2.3, you may supply your own built-ins, or even override the standard built-ins.)

> **Systems**. Some modules are vital to the behaviour of the system. Unless you are ready to help us tune the system, they need not concern you.
>
> **Booleans**. The module `BOOL` and the sort `Bool` have appeared frequently. Actually, `Bool` is declared in a submodule imported by `BOOL`. And `RWL` is a module that declares transition predicate "`_==>_`".
>
> **Numbers**. The system supports usual arithmetics within a usual number system. The modules involved are `NAT`, `INT`, `RAT` and `FLOAT`.
>
> **Strings**. `QID` (quoted identifier), `CHARACTER`, `STRING` define operations on characters and strings.
>
> **Pairs**. A simple pair, triple, and quadruple are defined in `2TUPLE`, `3TUPLE`, and `4TUPLE`.
>
> **Miscellaneous**. For historical reasons, `PROPC` (Section 7.7), `TRIV` (Section 8.2.1) are built-in modules.

There are also a set of library modules and examples in the distribution package.

Warning Elements in built-in sorts are defined as corresponding Lisp objects, and the system may compute them by stealth. For example, an expression `2.1e1` is an element of `Float`, and is *identified* with `21.0`; `2/1` is an element of `Rat`, and is identified with `2`.

Warning The built-ins and their definitions may change.

Chapter 12

Reference Manual

Here is a brief summary of lexical conventions and syntactic definitions used in CafeOBJ codes and commands.

12.1 Lexical Analysis

On the lexical surface a CafeOBJ code is a sequence of tokens and separators. A *token* is a sequence of printable ASCII characters (octal 40 through 176)[1], and is classified as either a self-terminating character, an identifier, or an operator symbol. Each class is explained below.

A *separator* is a blank character — space, vertical tab, horizontal tab, carriage return, newline, or from feed.

On the one hand, any mumber of separators may appear between tokens. On the other hand, one or more separators must appear between any two adjacent non-self-terminating tokens[2].

12.1.1 Reserved Words

There is *no* reserved word. You can use such keywords as `module` and `op` for identifiers or operator symbols.

12.1.2 Self-Terminating Characters

A self-terminating character is a printable ASCII character which by itself constitutes a token. There are seven such characters.

```
( ) , [ ] { }
```

12.1.3 Identifiers

An *ident* (identifier) is a sequence of any printable ASCII characters other than

self-terminating characters
`.` (period)
`"` (double quote)

[1]The current system accepts EUC coded Japanese characters also, but this is beyond the definition of the language.

[2]The same rule applies to terms. Further, if an operator symbol contains blanks or self-terminating characters, it is sometimes neccessary to enclose a term with such operator as top by parentheses for disambiguation.

The CafeOBJ reader is case-sensitive, and `Nat` and **nat** are different identifiers. *ident*s appear as module names, view names, parameter names, sort names, variables names, slot names, and labels.

12.1.4 Operator Symbols

An *operator_symbol* is a sequence of any ASCII characters, including non-printables, except EOT (control-D).

An underbar has a special meaning in an operator symbol. It reserves the place where an argument is inserted. A single underbar _ cannot be an operator symbol.

12.1.5 Comments

A *comment* is a sequence of characters which begins with one of the following four character sequences

```
--   -->
**   **>
```

and that ends with a newline character. In between, a comment may contain only printable ASCII characters and horizontal tabs.

Comments may act as separators, but their appearance is restricted (See the next section).

12.2 CafeOBJ Codes

A CafeOBJ code is a sequence of modules, views, and evaluation commands.

program ::= { *module* | *view* | *eval* } ∗

12.2.1 Modules

where a *module* is

```
module          ::= module_type module_name [list_of_parameters] [principal_sort]
                    "{" module_element * "}"
module_type     ::= module | module! | module*
module_name     ::= ident
parameters      ::= "(" parameter +, ")"
parameter       ::= [ protecting | extending ] paramter_name "::" module_expression
parameter_name  ::= ident
principal_sort  ::= principal-sort sort_name

module_element  ::= import | sort | record | operator | variable | axiom | comment
import          ::= { protecting | extending | using } "(" module_expression ")"
sort            ::= visible_sort | hidden_sort
visible_sort    ::= "[" sort_decl +, "]"
hidden_sort     ::= "*[" sort_decl +, "]*"
sort_decl       ::= sort_name + [ supersorts + ]
supersorts      ::= "<" sort_name +
sort_name       ::= ident[qualifier] [3]
qualifier       ::= "."module_expression[qualifier] [4]
record          ::= record sort_name [ upper_record + ] "{" { slot | comment } + "}"
upper_record    ::= "[" sort_name [ "(" slot_rename +, ")" ] "]"
slot_rename     ::= slot_name "->" slot_name
slot_name       ::= ident
slot            ::= { slot_name ":" sort_name }
                  | { slot_name "=" "("term")" ":" sort_name }
operator        ::= { "op" | "bop" } operator_symbol ":" arity "->" coarity [ attribute_list ]
arity           ::= sort_name *
coarity         ::= sort_name
attribute_list  ::= "{" attribute * "}"
attribute       ::= constr | associative | commutative | idempotent | identity
                  | strategy | precedence | l-assoc | r-assoc | coherent
identity        ::= { id: | idr: } term
strategy        ::= strat: "(" integer + ")" [5]
precedence      ::= prec: natural_number [6]
variable        ::= { var variable_name ":" sort_name }
                  | { vars variable_name + ":" sort_name }
variable_name   ::= ident
axiom           ::= equation | cequation | transition | ctransition
equation        ::= { eq | beq } [ label ] term "=" term "."
label           ::= "[" ident "]" ":"
cequation       ::= { ceq | bceq } [ label ] term "=" term if term "."
transition      ::= { trans | btrans } [ label ] term "=>" term "."
ctransition     ::= { ctrans | bctrans } [ label ] term "=>" term if term "."
```

12.2.2 Views

and a *view* is

```
view            ::= view view_name from module_expression to module_expression
                    "{" view_element *, "}"
view_name       ::= ident

view_element    ::= sort_map | view_op_map | variable
sort_map        ::= { sort | hsort } sort_name "->" sort_name
view_op_map     ::= op_map | term_map
op_map          ::= { op | bop } op_name "->" operator_symbol
op_name         ::= operator_symbol | "(" operator_symbol ")"qualifier
term_map        ::= term "->" term
```

12.2.3 Evaluation Commands

and an *eval* is

eval	::=	*eval_interpreter* | *eval_tram*
eval_interpreter	::=	{ **reduce** | **execute** | **breduce** } [*context*] *term* "**.**"
context	::=	**in** *module_expression* "**:**"
eval_tram	::=	**tram** { **reduce** | **execute** } [*context*] *term* "**.**"

12.2.4 Module Expressions

and an *module_expression* is

module_expression	::=	*module_name* | *sum* | *rename* | *instantiation* | "**(**" *module_expression* "**)**"
sum	::=	*module_expression* { "**+**" *module_expression* } +
rename	::=	*module_expression* "*****" "**{**" *rename_map* +, "**}**"
rename_map	::=	*sort_map* | *op_map*
instantiation	::=	*module_expression* "**(**" { *parameter_name*[*qualifier*] "**<=**" *aview* } + "**)**"
aview	::=	*view_name* | *module_expression*
	—	{ **view to** *module_expression* "**{**" *view_element* *, "**}**"

When a module expression is not fully parenthesized, the proper nesting of subexpressions may be ambiguous. The following precedence rule is used to resolve such ambiguity:

sum < *rename* < *instantiation*

12.2.5 Terms

A *term* is a sequence of any ASCII characters except EOT.

12.2.6 Abbreviations

The official abbreviations are listed below.

`associative`	`assoc`
`bceq`	`bcq`
`bctrans`	`bctrns`
`breduce`	`bred`
`btrans`	`btrns`
`ceq`	`cq`
`commutative`	`comm`
`ctrans`	`ctrns`
`execute`	`exec`
`extending`	`ex`
`idempotent`	`idem`
`module`	`mod`
`module!`	`mod!`
`module*`	`mod*`
`principal-sort`	`psort`
`protecting`	`pr`
`reduce`	`red`
`trans`	`trns`
`using`	`us`

12.2.7 Sugar Coatings

Simultaneous Operator Declaration. Several operators with the same arity, coarity, and operator attributes can be declared at once by `ops`. The syntax is

operators ::= { `ops` | `bops` } *operator_symbol* + ": " *arity* "->" *coarity* [*attribute_list*]

and the effect is the same as many *operator*s as there are *operator_symbols*s.

In simultaneous declarations, parentheses are sometimes necessary to separate operator symbols. They are always required if an operator symbol contains periods, blanks or underbars.

Predicate. A *predicate* is a syntactic sugar for operator declarations with arity `Bool`, and is defined as

predicate ::= { `pred` | `bpred` } *operator_symbol* ": " *arity* [*attribute_list*] [7]

The above definition is a specialisation of *operator*, as

::= { `op` | `bop` } *operator_symbol* ": " *arity* "->" `Bool` [*attribute_list*]

Blocks of Declarations. Importations, signatures and axioms can be clusterd into blocks *imports*, *signature*, and *axioms* respectively.

imports ::= `imports` "{" { *import* | *comment* } + "}"
signature ::= `signature` "{" { *sort* | *record* | *operator* | *comment* } + "}"
axioms ::= `axioms` "{" { *variable* | *axiom* | *comment* } + "}"

Views. It is possible to identify parameters by positions, not by names. For example, if a parametric module is declared as

```
module! OYVEH (A1 :: TH1, A2 :: TH2) { ... }
```

the form

```
OYVEH (V1, V2)
```

is equivalent to

```
OYVEH (A1 <= V1, A2 <= V2)
```

Moreover, "`view to`" construct always be omitted. So

```
OYVEH (A1 <= view to M {...})
```

can be written as

```
OYVEH (A1 <= M {...})
```

12.3 Command Summary

Following is the top level commands that do not constitute CafeOBJ codes.

command	::=	*show* | *describe* | *set* | *input* | *check* | *regularise* | *save* | *restore* | *save_system*
	|	*protect* | *unprotect* | *clean_memo* | *reset* | *full_reset* | *parse* | *lisp_call* | *select* | *open*
	|	*close* | *start* | *apply* | *match* | *choose* | *find* | *provide* | *require* | *autoload*
	|	*eof* | *prompt* | *quit* | *cd* | *ls* | *pwd* | *help* | *shell_call* | *dribble* | *period* | *compile*
show	::=	**show** { *help* | *module_contents* | *switches* }
describe	::=	**describe** { *help* } | *module_contents* }
set	::=	**set** { *set_help* | *set_switch* | *set_other* }
input	::=	**input** *file_name*
protect	::=	**protect** *module_name*
unprotect	::=	**unprotect** *module_name*
full_reset	::=	**full-reset** | { **full reset** }
open	::=	**open** *module_expression*
close	::=	**close**
start	::=	**start** *term* "."
parse	::=	**parse** [*context*] *term* "."
cd	::=	**cd** *file_name*
quit	::=	**quit**
period	::=	"."
shell_call	::=	"!" *shell_command*
provide	::=	**provide** *feature*
require	::=	**require** *feature* [*file_name*]
ls	::=	**ls** *file_name*
compile	::=	**tram compile** [*compile_option*] *module_expression*
compile_option	::=	**-exec**
eof	::=	**eof**
save	::=	**save** *file_name*
restore	::=	**restore** *file_name*
clean_memo	::=	**clean memo**
dribble	::=	**dribble** *file_name*
select	::=	**select** *module_expresion*
lisp_call	::=	{ **lisp** | **lispq** } *S_expression*
help	::=	"?"
autoload	::=	**autoload** *proc_name*
find	::=	**find** { **rule** | **+rule** | **-rule** }
match	::=	**match** *term_specifier* **to** *match_pattern* "."
term_specifier	::=	**top** | **term** | **it** | *subterm* | "(" *term* ")"
match_pattern	::=	{ [**all**] { **rules** | **+rules** | **-rules** } }
	|	*term*
apply	::=	**apply** *action* [*substitution*] *range selection*
select	::=	
check	::=	
regularise	::=	**regularize** *module_name*

Appendix A

A Brief Guide to Basic Notions and Terminology

Bibliography

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

[2] Diaconescu, R. and Futatsugi, K., *Logical Semantics of CafeOBJ*, Technical Report IS-RR-96-0024S, Japan Advanced Institute for Science and Teleology, 1996

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

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

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

[6] Goguen, J. and Diaconescu, R., "An Oxford Survey of Order Sorted Algebra", Mathematical Structures in Computer Science, Vol.4, 1994, pp.363–392

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

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

[9] Goguen, J., Winkler, T., Meseguer, J., Futatsugi, K., and Jouannaud, J.-P., *Introducing OBJ*, Technichal Report SRI-CSL-92-03, SRI International, 1992

[10] Jacobs, B. and Rutten, J., "A Tutorial on (Co)Algebras and (Co)Induction", *EATCS Bulletin*, No.62, EATCS, 1997, pp.222–259

[11] Klop, J.W., "Term Rewriting Systems: A Tutorial", *EATCS Bulletin*, No.32, EATCS, 1987, pp.143–182

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

[13] Meseguer,J. and Goguen,J.A., "Initiality, induction and computability", *Algebraic Methods in Semantics*, Cambridge University Press, 1984, pp.459–541

Index