System Modules

Denys Duchier
Leif Kornstaedt
Martin Homik
Tobias Müller
Christian Schulte
Peter Van Roy

Version 1.3.2
June 15, 2006

Abstract

The Mozart system consists of two complementary parts: first comes the Oz *core* language which lays the semantic foundations and whose programmatic interface is documented in *“The Oz Base Environment”*, then come all the extras necessary for practical programming; these are documented here.

Credits

Mozart logo by Christian Lindig

License Agreement

Contents

Part I

Application Programming

1

Application Support: `Application`

1.1 The `Application` Module

The `Application` module provides procedures for accessing the application's arguments, and for terminating applications.

`getCgiArgs`

```
{Application.getCgiArgs +Spec ?R}
```

acquires the arguments (both `GET` and `POST` methods supported) and parses them according to `Spec` as described in Section 1.3. Returns the options in `R`.

`getCmdArgs`

```
{Application.getCmdArgs +Spec ?R}
```

acquires the arguments from the system property `'application.args'` and parses them according to `Spec` as described in Section 1.3. Returns the options in `R`.

`getGuiArgs`

```
{Application.getGuiArgs +Spec ?R}
```

pops up a graphical interface with which the user can interactively and comfortably edit the possible options described by `Spec` as described in Section 1.3. Returns the options in `R`.

`getArgs`

```
{Application.getArgs +Spec ?R}
```

This is the recommended way of acquiring an application's arguments. It invokes either `Application.getCmdArgs` or `Application.getGuiArgs` depending on the value of boolean property `'application.gui'`. The latter is set to **`true`** when the `ozengine` is invoked with option `-gui`.

`processArgv`

```
{Application.processArgv +Spec ?Ss}
```

performs argument parsing on the explicitly given list of strings `Ss` according to specification `Spec`.

`processCgiString`

```
{Application.processCgiString +Spec ?S}
```

performs argument parsing on the explicitly given CGI query string `S` according to specification `Spec`.

Error Handling If an error is encountered in the input, an error exception of the form `ap(usage VS)` is raised. VS describes the error in textual form.

`exit`

```
{Application.exit +I}
```

terminates the application with return status `I`, `0` indicating success and non-`0` indicating failure of some kind.

1.2 Parsing Conventions

This section describes how the arguments are acquired and what basic syntax is used for parsing them.

1.2.1 Parsing of CGI Arguments

The CGI always passes arguments as name/value pairs, where the name is separated from the value by an equals sign and the individual pairs are separated by ampersands.

Boolean options *option* may be given as *option*`=yes` or *option*`=no`. Option names may be abbreviated, as long as they remain unambiguous.

1.2.2 Parsing of Command Lines

We distinguish between long option names and single character options. Long options are given as `-`*option* or `-`*option*`=`*value*; option names may be abbreviated as long as they remain unambiguous. Single-character options are given as `-`*x*, eventually followed by a value. Several single-character options may be combined, e.g., `-xy` means `-x -y` (provided `x` does not take an argument). The argument to a single-character option may be attached to the option character, i.e., you can write `-`*xvalue* or `-`*x value*.

Boolean options *option* may be given as `-`*option* (meaning **`true`**) or `-no`*option* (meaning **`false`**). A single hyphen `-` by itself is not considered to be an option and thus is returned unchanged. Parsing stops at a double hyphen `-` appearing by itself; the double hyphen itself does not appear in the output.

In the case of an unrecognized long option name or single-character option, or of an ambiguous long option prefix, an error is raised.

1.3 Option Specifications

There are several ways to specify the way the arguments are parsed; we present them in order of increasing processing power.

1.3.1 Plain

Syntax Specification The `plain` way of command line processing actually involves no processing at all. In CGI parsing, not even escaped characters are translated.

$\langle$spec$\rangle$ ::= `plain`

Returned Result For CGI scripts, the result consists of a list of pairs of strings (the name/value pairs), whereas for command lines, it consists of a list of strings.

1.3.2 List

Syntax Specification The `list` way of processing command line arguments takes care of determining what is a command line option, whether it takes a value, how its value to be interpreted, etc.

⟨spec⟩ += `list(`[`mode:` ⟨mode⟩] ⟨option⟩ … ⟨option⟩`)`

Using the `mode` specification, the command line parser can either be instructed to stop at the first non-option argument it encounters (`start`) or it can look for options on the whole command line (`anywhere`). The latter is the default if no `mode` is given.

⟨mode⟩ ::= `start` | `anywhere`

The integer fields of the option specification describe the individual options. An option must as least have an ⟨option name⟩. Furthermore, it may either be an alias for another option (if `alias` is given) or it may be a 'real' option actually visible to the application. Aliases are never returned to the application; they are always replaced by the option they stand for.

⟨option⟩ ::= ⟨option name⟩`(`[`char:` ⟨char or chars⟩] [`type:` ⟨type⟩]`)`
| ⟨option name⟩`(`[`char:` ⟨char or chars⟩] `alias:` ⟨alias⟩`)`

⟨option name⟩ ::= ⟨atom⟩

As mentioned in Section 1.2.2, options may be notated using single-character short forms. With the `char` specification one or several single-character short forms may be assigned to an option.

⟨char or chars⟩ ::= ⟨char⟩ | `[`⟨char⟩`]`

If no `type` is given, then the option does not take an argument. (Note that **`true`** will be used as the associated value in this case.) Boolean options have a special status, as has already been described in Section 1.2. The remaining type specifications, however, require an additional argument. The `list(`⟨primary type⟩`)` annotation interprets its argument as a comma-separated list of elements of a specific type.

⟨type⟩ ::= `bool`
| ⟨primary type⟩
| `list(`⟨primary type⟩`)`

There are four supported basic types and a 'generic' type. Integer and float arguments have to be given in Oz concrete syntax (with the exception that the unary minus sign may be notated as `-`); minimum and maximum values may also be specified. For arguments to be returned as atoms, a set of allowed values may be specified. Strings are returned as-is.

The generic type simply consists of a binary procedure with the signature `{P +S X}` which may arbitrarily transform the argument, given as a string.

```
⟨primary type⟩  ::=  int([min: ⟨int⟩] [max: ⟨int⟩])
                  |  float([min: ⟨float⟩] [max: ⟨float⟩])
                  |  atom([⟨atom⟩ ... ⟨atom⟩])
                  |  string
                  |  ⟨procedure⟩
```

Two different forms of alias are supported. Option name aliases simply state that this option name is equivalent to some other option name; the other option's argument description will be used for parsing this option as well. The second kind of alias states that this option is equivalent to another option used with the supplied value (or a combination of several options). In the latter case, the value will be transferred to the output without any additional transformations.

```
⟨alias⟩  ::=  ⟨option name⟩
           |  ⟨option name⟩#⟨value⟩
           |  [⟨option name⟩#⟨value⟩]
```

Returned Result The result of this processing step is a list of parsed options, interspersed with non-parsed arguments, a so-called ⟨option list⟩. All option names in this list are the canonical (i.e., not aliased and unabbreviated) forms. The list respects the order in which the arguments were given.

```
⟨option list⟩  ::=  [⟨arg or option⟩]

⟨arg or option⟩  ::=  ⟨option name⟩#⟨value⟩
                   |  ⟨string⟩
```

List Examples

The following will give some examples taken from the `ozc` program, which is the Oz command-line compiler.

Basics The `ozc` program has a command line option to tell it to output usage information. The easiest way to specify such an option is:

```
help
```

Furthermore, we want to support a popular single-character abbreviation for it:

```
help(char: &h)
```

We might even support several single-character abbreviations for convenience.

```
help(char: [&h &?])
```

We can now write any of `-help`, `-h`, or `-?` for this option. (We might also abbreviate the long form to one of `-h`, `-he`, or `-hel`, provided that this would still be unambiguous.) The returned option list would be `[help#true]`.

Boolean Options There is another option to tell `ozc` to be verbose about what it is doing. Say we specified it as follows:

```
verbose(char: &v)
```

This means that we can write `-verbose` or `-v`. In contrast, the following specification additionally allows for `-noverbose`:

```
verbose(char: &v type: bool)
```

Aliases Some people prefer to write `-noverbose` as `-quiet`, so we introduce an alias for it:

```
quiet(char: &q alias: verbose#false)
```

This is an alias with associated value. In contrast, the following option (not supported by `ozc`, by the way) would be an alias without value:

```
gossipy(alias: verbose)
```

This would allow us to write `-gossipy` for `-verbose` and `-nogossipy` for `-noverbose`.

String Arguments The following example illustrates another type of argument than boolean:

```
include(type: string)
```

Saying `-include=x.oz` for instance would tell `ozc` to load and compile this file before tackling its ‘real’ job. Together with the following option, we get an example for when the order beween different arguments may matter:

```
'define'(char: &D type: atom)
```

Saying

```
--define=MYMACRO1 --include=x.oz --include=y.oz
```

for instance would mean that `MYMACRO1` would be defined for both `x.oz` and `y.oz`, whereas in

```
--include=x.oz --define=MYMACRO1 --include=y.oz
```

it is only defined for `y.oz`.

By the way, this option has a single-character abbreviation *and* an explicit argument (in contrast to the implicit boolean arguments above): We can thus write either `-D MYMACRO1` or `-DMYMACRO1` instead of `-define=MYMACRO1`.

List Arguments In `ozc`, actually, a list of macro names is allowed for this option:

```
'define'(char: &D type: list(atom))
```

This also supports, e.g., `-DMYMACRO1,YOURMACRO17 x.oz`. This would return the option list `['define'#['MYMACRO1' 'YOURMACRO17'] "x.oz"]`.

Range Limitations Sometimes one wants to limit the range of allowed values:

```
compress(char: &z type: int(min: 0 max: 9))
```

This would allow us to write `-z9`, but not `-z17`. For atom arguments, sometimes only a limited set of values is sensible:

```
mode(type: atom(help core outputcode
                feedtoemulator dump executable)
```

For these, `ozc` also provides the better known aliases such as:

```
dump(char: &c alias: mode#dump)
```

1.3.3 Record

Syntax Specification The additional processing step involved in `record` kind specifications is that additional contextual conditions may be checked, and the result is returned in a different form.

Basically, the `record` specification is a strict extension of the `list` specification.

⟨spec⟩ += `record(`[`mode:` ⟨mode⟩] ⟨option⟩ … ⟨option⟩`)`

The specifications for 'real' (i.e., non-alias) options take some more information into consideration, namely how often the option may appear and how several occurrences combine (⟨occ⟩), and whether it is a required option (`optional`; the default is **`true`**) or whether it takes a default value (`default`), which it does not by default. At most one of `default` and `optional` may be given.

⟨option⟩ += ⟨option name⟩`(`
  `1:` ⟨occ⟩
  [`char:` ⟨char or chars⟩]
  [`type:` ⟨type⟩]
  [`default:` ⟨value⟩ | `optional:` ⟨bool⟩]`)`

An option may be allowed to occur at most once (`single`) or any number of times. In the latter case, the result may either respect all occurrences (`multiple`), or it may ignore all but the first (`leftmost`) or last (`rightmost`) occurrence. When all occurrences are respected, a list of them (preserving the order) is returned.

⟨occ⟩ ::= `single` | `multiple` | `leftmost` | `rightmost` | `accumulate(P)`

When `accumulate(P)` is specified, procedure `P` is called for each occurrence of the option. It takes two arguments: the option (as an atom) and the parsed value. This can be used to accumulate multiple occurrences of related options into one list. See, for example, options `-include` and `-exclude` of the Oz linker `ozl`[1].

[1]Chapter *The Oz Linker:* `ozl`, *(Oz Shell Utilities)*

Returned Result The result consists of an option record. All options which had an explicit ⟨occ⟩ given in their specification are moved from the option list into this record, the feature being the option name, the subtree the associated value. Defaulted options that have not been overridden by the argument list appear in this record with their default value. Only optional options may be missing from this record, namely when they have not been specified in the argument list. Those options which did not have an explicit ⟨occ⟩ given in their specification are found, interspersed with non-parsed arguments, in an option list under feature `1` of the option record.

```
⟨option record⟩  ::=  optRec(1: ⟨option list⟩
                                ⟨option name⟩: ⟨value⟩ ...
                                ⟨option name⟩: ⟨value⟩)
```

Record Examples

Naturally, all examples given for `list` are also valid for `record`, but in order to make them appear in the resulting option record, we have to specify some additional things. This section illustrates this.

Basics For example, with the `mode` as specified earlier, the argument list `-mode=dump` would result in the following option record:

```
optRec(1: [mode#dump])
```

In order to make it appear, we add the keyword `single` to the specification, stating at the same time that this option can be given at most once:

```
mode(single
     type: atom(help core outputcode
                feedtoemulator dump executable))
```

Then the option record for `-mode=dump` would look like this:

```
optRec(1: nil mode: dump)
```

Default or Required Since the `mode` gives the basic mode of operation for `ozc`, we would be lost if was not given in the arguments, because it would not appear in the option record. To enforce its presence, we can either supply a default:

```
mode(single type: atom(...) default: feedtoemulator)
```

or make it a required option:

```
mode(single type: atom(...) optional: false)
```

Multiple Occurrences The keyword `single` stated that an option may appear at most once in the option record. For some options, this in inadequate. If we want an option to be allowed to occur multiply in the argument list, we have to specify what this means. For instance,

```
verbose(rightmost char: &v type: bool)
```

means that all but the last occurrences of `verbose` are ignored. By the way, in `ozc`, `verbose` actually has a non-boolean default:

```
verbose(rightmost char: &v type: bool default: auto)
```

This allows for three modes of operation: The default is to only output messages if they are 'interesting'. When being `-verbose`, also uninteresting messages are output, whereas being `-quiet`, even the interesting messages are suppressed.

Collecting in Lists It is also possible to state that one wished all occurrences of the same option to be collected in a list. This does not occur in `ozc`, so we give a fictitious example here:

```
cattle(multiple type: list(atom) default: nil)
```

Giving this argument several times, say, `-cattle=angus,belgianred`, `-cattle=charolais` and `-cattle=dexter,highland` on one command line would result in the following option record:

```
optRec(1: nil
       cattle: [angus belgianred charolais dexter highland])
```

2

Module Managers: `Module`

Module managers grant access to modules identified by urls. Examples and more information on module managers can be found in *“Application Programming”*.

2.1 Basics

A module manager maintains a module table, that maps urls to modules. To be more precise, the table maps a url to a module or to a future for a module (this is explained later).

A module manager supports the following operations:

link

Linking takes a url U as input and returns a module M or a future F for a module.

We say that M or F is linked from U. Most of the time we will not distinguish between M and F.

Depending on whether the module table already contains an entry for M, the following happens:

1. If the module table already contains an entry M for the url U, linking just returns M.
2. If no entry for U is available, a new future F is created and stored under U in the module table. F is returned.

   As soon as the value for the future F is requested, a new thread is created that installs a module from the url U. This is called dynamic linking or linking on demand.

   If under the url U a pickled functor is stored, the module is computed by first loading the functor G. Then G is applied with respect to the url U (this is explained later) which yields a module to which the future F is bound. This also explains why it is okay to not make a distinction between module and future for a module, since the distinction does not has any consequences as it comes to module access.

   If the url U refers to a system module (see also Chapter *Module Managers, (Application Programming)*) the system module is returned.

   The url U can also refer to a native functor, this is described in detail in Part *Native C/C++ Extensions, (Application Programming)*.

apply Application takes a functor F and a base url U and returns a module M.

First the import urls of F are resolved with respect to the base url U. Then the resolved urls are used for linking. The returned modules are called argument modules. Then the body of the functor is applied to the argument modules.

enter Entering takes a url U and a module M as input.

The module M is added to the module table under U. If the module table already contains an entry for U, an exception is raised.

A module manager is implemented as an instance of the class `Module.manager`. The class provides methods to link and apply functors and to enter modules into the module manager's module table.

2.2 Module Names and URLs

As has been explained above, each module is refered to by a url U, some of which are Oz specific in that they refer to system modules. We just say that the module *has the url* U.

A *module name* is a shortcut for a module url. The mapping of module names to full urls is explained in detail in Chapter *Module Managers, (Application Programming)*.

2.3 The Class `Module.manager`

Module managers are created as instances of the class `Module.manager`. For predefined abstractions that are build on top of module managers see Section 2.4.

`init`

```
init()
```

Initializes the module manager.

`link`

```
link(url:+UrlV ModuleR <= _)
link(name:+NameV ModuleR <= _)
```

Links the module identified either by a url `UrlV` (a virtual string) or a module name `NameV` (a virtual string). Returns the module `ModuleR` (which might be a future to a module).

The argument for the module is optional, if it is omitted the module is requested immediately.

`apply`

```
apply(+Functor ModuleR <= _)
apply(url:+UrlV +Functor ModuleR <= _)
apply(name:+NameV +Functor ModuleR <= _)
```

Applies the functor `Functor`, where the url `UrlV` (a virtual string) or the module name `NameV` (a virtual string) serve as base url for linking the functor's import. If neither a module name nor a url is given, the current working directory is taken as base url.

The argument for the module is optional.

Please note that the resulting module is *not* added to the module table, the URL argument only serves as base url for the functor's import.

`enter`

```
enter(url:+UrlV ModuleR)
enter(name:+NameV ModuleR)
```

Installs the module `ModuleR` under the url `UrlV` (a virtual string) or the module name `NameV` (a virtual string).

Raises an exception if the module manager already has a module under that particular url installed.

2.4 Predefined Abstractions

`link`

```
{Module.link +UrlVs Rs}
```

Takes a list `UrlVs` of urls (as virtual strings) and and returns the list of modules created by linking.

All functors are linked by the same module manager, however each application of `Module.link` employs a new module manager. This has the following consequences:

- Modules imported by several functors are shared.
- Each application of `Module.link` links required functors anew. That is, after replacing a functor on the file system, an application of `Module.link` considers the new functor for linking.

`Module.link` is defined as follows:

```
fun {Module.link UrlVs}
   ModMan = {New Module.manager init}
in
   {Map UrlVs fun {$ Url}
                 {ModMan link(url:Url $)}
              end}
end
```

`apply`

```
{Module.apply +Xs Rs}
```

Takes a list of functors or pairs of urls (as virtual strings) as input. The url in a pair of url and functor describes the base url with which the import urls of the functor gets resolved. If it is missing the current working directory is used for url resolution.

Returns a list of modules computed by functor application.

`Module.apply` is defined as follows:

```
fun {Module.apply UFs}
   ModMan = {New Module.manager init}
in
   {Map UFs fun {$ UF}
               case UF of U#F then
                  {ModMan apply(url:U F $)}
               else
                  {ModMan apply(UF $)}
               end
            end}
end
```

Part II

Constraint Programming

3

Constraints-Specific Type Structure and Modes

This section presents those types and modes which are specific for the constraint extensions.

3.1 Type Structure

There are two additional secondary types.

Vector A vector is a record with a label different from `'|'` or a list. The elements of the list or the fields of the record are called the elements of the vector. A finite domain vector is a vector all of whose elements are finite domain integers.

Specification of Sets of Integers A specification of sets of integers `Spec` is used in cointext of finite domain and finite set constraints. It is recursively defined as follows.

Spec	::=	*simpl_spec*
	|	compl(*simpl_spec*)

simpl_spec	::=	*range_descr*
	|	[*range_descr*+]
	|	`nil`

range_descr	::=	*integer*
	|	*integer*`#`*integer*

integer	::=	`FD.inf`,...,`FD.sup`
	|	`FS.inf`,...,`FS.sup`

A specification of sets of integers denotes a set of integers which is either the union of integer singletons *i* and integer intervals *i*#*i*, or the complement `compl(...)` of such a set relative to $\{$`FD.inf`$,\ldots,$`FD.sup`$\}$ resp. $\{$`FS.inf`$,\ldots,$`FS.sup`$\}$. Note that an empty set is specified by `nil`.

In context of finite domain constraints for example, `2#5` denotes the set $\{2,\ldots,5\}$, the specification `[1 10#20]` denotes the set $\{1,10,\ldots,20\}$, and `compl(2#5)` denotes $\{0,1,6,\ldots,$`FD.sup`$\}$.

The value of `FD.inf` and `FS.inf` is 0 and the value of `FD.sup` and `FS.sup` is 134217726. These values are implementation-dependent.

Weight Specifications Weight specifications `SpecW` occur in conjunction with set constraints (see Section 7.9) and are defined as follows.

SpecW	::=	`nil`
	|	[*ElemDescr*]+

ElemDescr	::=	*Int*`#`*Int*
	|	`(`*Int*`#`*Int*`)#`*Int*
	|	`default#`*Int*

3.2 Signatures

Types

The additional type abbreviations are listed in Figure Figure 3.1.

Figure 3.1 Type Abbreviations

Abbreviation	Type
D	finite domain integer
M	finite set of integers
Xr	records of type *X*
Xt	tuples of type *X*
Xv	vectors of type *X*
Xvv	vectors of vectors of type *X*
Xrr	records of records of type *X*

Modes

Given a constraint store, every variable is in exactly one of the following three states. It is *free* if the store knows nothing about the variable apart from equalities, *determined* if the store knows the top-level constructor, and *kinded* if the variable is neither free nor determined. Variables which are either determined or kinded are called *constrained*.

The base language does not allow to constrain a variable without determining it. Most procedures of the base language wait until their arguments are determined.

Input Modes $*$, $+$ In the constraint extension, a variable can be constrained before it becomes determined. Accordingly, the constraint extensions use additional input modes $*$ and $ which synchronize more weakly than $+$. The application of a procedure `P` waits until its inputs ($+$, $*$) are determined or constrained, respectively. If the input arguments are well-typed, `P` returns outputs of the specified types. Ill-typed input arguments produce a runtime type error (on completion of `P`).

Propagators Note that it is perfectly possible that an input argument is constrained further. This is the case for many propagators, which have the following typical moding.

```
{P *X *Y *Z}
```

Note also that modes only partially specify the synchronisation behavior of a procedure.

Nestable Input Mode $ The mode $ slightly weakens $*$ to allow for nesting of propagators. When *n* arguments of a propagator have input mode $, then this propagator waits until $n-1$ of them are constrained and then it constrains the remaining *n*th argument according to its type.

3.2.1 Notational Conventions

Notational conventions are explained in context of finite domain constraints but apply of course for finite set constraints too.

Specification Input The signature

```
{FD.int +Spec ?D}
```

specifies that an application of `FD.int` waits until +`Spec` is *ground*, i.e., contains no free variables. Arguments of the form +`Spec` never occur. The signature

```
{FD.distinct *Dv}
```

specifies that an application of `FD.distinct` waits until its argument `Dv` is determined and all its elements are constrained to finite domain integers. Analogously, +`Iv` specifies that the application waits until `Iv` and all its elements are determined. The scheme

```
{FD.sumCN +Iv *Dvv  +A  *D}
```

specifies that the application waits until *`Dvv` and all its elements are determined, and until their elements are constrained to finite domain integers.

Generic Propagators For some procedures like that for generic propagators, an atom occurring as an argument denotes a relation symbol. For example,

```
{FD.sum [X Y Z] '=:' D}
```

denotes the constraint

$$X + Y + Z = D$$

If A is the atomic argument, $\sim_A$ is the corresponding arithmetic relation. For A the atoms `'=:'`, `'>:'`, `'>=:'`, `'<:'`, `'=<:'`, and `'\\=:'` are allowed. The relations are $=$, $>$, $\geq$, $<$, $\leq$, and $\neq$, respectively.

4

Search Engines: `Search`

This chapter describes various search engines. The engines fall into the following categories.

Basic Search Engines Easy to use engines for single, all, and best solution search.

General Purpose Search Engines Engines that offer additional support for:

Recomputation Recomputation allows to trade space for time, allowing to solve problems which otherwise would use too much memory.

Killing The execution of engines can be killed.

Output Solutions computed can be returned as procedures or first-class computation spaces.

Search Object (page 27) The search object supports demand-driven search for single, all, and best solutions. Search can be stopped and resumed as needed. The object supports recomputation and the different kinds of output as described above.

Parallel Search Engines (page 28) Parallel search engines use multiple networked computers to speed up the exploration of search trees. During exploration of a search tree entire subtrees are delegated to Oz engines that run on different computers in parallel.

Oz Explorer Besides of the engines described here, Mozart features the OzExplorer, an interactive graphical search engine. A short description of its use can be found in Section *The Explorer*, *(Finite Domain Constraint Programming in Oz. A Tutorial.)*. Reference information on the Oz Explorer can be found in *"Oz Explorer – Visual Constraint Programming Support"*, a research paper is [9].

4.1 Basic Search Engines

All these engines take a script as input and return a list of its solutions.

`base.one`

```
{Search.base.one +ScriptP ?Xs}
```

returns a singleton list containing the first solution of the script +ScriptP (a unary procedure) obtained by depth-first search. If no solution exists, nil is returned.

As an example,

```
{Search.base.one proc {$ X}
                    choice
                       choice X=ape [] X=bear end
                    [] X=cat
                    end
                 end}
```

returns the list [ape].

base.all

```
{Search.base.all +ScriptP ?Xs}
```

returns the list of all solutions of the script +ScriptP (a unary procedure) obtained by depth-first serach. As an example,

```
{Search.base.all proc {$ X}
                    choice
                       choice X=ape [] X=bear end
                    [] X=cat
                    end
                 end}
```

returns the list [ape bear cat].

Search.base.best

```
{Search.base.best +ScriptP +OrderP ?Xs}
```

returns a singleton list containing the best solution with respect to the order +OrderP (a binary procedure) of the script +ScriptP (a unary procedure) obtained by branch and bound search. If no solution does exist, nil is returned.

The branch and bound strategy works as follows. When a solution is found, all the remaining alternatives are constrained to be *better* with respect to the order +OrderP. The binary procedure +OrderP is applied with its first argument being the previous solution, and its second argument the root variable of a space for one of the remaining alternatives.

For instance, the following script constrains its root variable to a pair of integers, such that a certain equation holds between its components.

```
proc {Script Root}
   X={FD.int 1#10} Y={FD.int 1#10}
in
   Y =: 10 - X - 2*Y
   Root = X#Y
   {FD.distribute split Root}
end
```

With the order

```
proc {MaxSum Old New}
   Old.1 + Old.2 <: New.1 + New.2
end
```

we can search for a solution with maximal sum of `X` and `Y` by

```
{SearchBest Script MaxSum}
```

This returns the singleton list `[7#1]`.

Similarly, we can search for the solution with the maximal product, by using the order:

```
proc {MaxProduct Old New}
   Old.1 * Old.2 <: New.1 * New.2
end
```

in:

```
{SearchBest Script MaxProduct}
```

This returns the singleton list `[4#2]`.

4.2 General Purpose Search Engines

This section describes the search engines found in the module `Search`. All of these engines support recomputation, the possibility to stop their execution and various kinds of output.

Recomputation. Scripts which create a large number of variables or propagators or scripts for which the search tree is very deep might use too much memory to be feasible. The search engines described in this section feature support for so-called recomputation. Recomputation reduces the space requirements for these scripts in that it trades space for time.

Search engines that do not use recomputation, create a copy of a computation space in each distribution step. This copy is needed such that the engine is able to follow more than one alternative of a choice.

If, for instance, a single solution search engine finds a solution after 200 distribution steps (i.e. the search tree has a depth of 201), 200 copies are created and stored by the engine.

Recomputation reduces the number of copies needed: Instead of creating a copy in each distribution step, only every n-th distribution step a copy is created. A space for which no copy has been created can be recomputed from a copy located higher above in the search tree by recomputing some distribution steps. In the worst case, $n-1$ distribution steps have to be recomputed. The parameter n is the so-called recomputation distance. A recomputation distance of n means that the *space* needed *decreases* by a factor of n and that the *time* needed *increases* by a factor of n.

The following search engines take the recomputation distance as an argument (it is denoted by *RcdI*). A value of `2` for *RcdI* means that only each second distribution step a copy is created. The value `1` for *RcdI* means that in each distrbution step a copy is created, that is no recomputation is used. Values less than `1` mean that none but an initial copy is created: from this initial copy all other spaces are recomputed.

Recomputation can also *reduce* both *space and time* requirements. Searching a single solution of a script which features a good heuristic (i.e. there are only very few failures) creates copies which are not used. Recomputation avoids this, resulting in improvement with respect to both space and time.

Recomputation requires that the distribution strategy used in the script be *deterministic*. Deterministic means that the created choices and their order are identical in repeated runs of the script. This is true for all strategies in the finite domain module, but for example not for strategies with randomized components.

Killing the Engine. All engines described in this section return a nullary procedure, which is denoted by +`KillP`. Applying this procedure kills the search engine.

A search engine, which can be stopped and resumed is described in Section Section 4.3.

Different Types of Output. Each of the engines is provided with three different types of output. The first kind returns a list of solutions as the engines in Section 4.1. The second kind returns a list of unary procedures. Applying one of these procedures merges a copy of the succeeded space and gives reference to its root variable variable by the actual argument of the procedure application. The third kind returns a list of succeeded spaces.

4.2.1 Single Solution Search

`one.depth`

```
{Search.one.depth +ScriptP +RcdI
                  ?KillP ?Xs}
```

returns a singleton list containing the first solution of the script +`ScriptP` (a unary procedure) obtained by depth-first search. If no solution exists, `nil` is returned.

For instance, the procedure `Search.base.one` (see Section 4.1) can be defined as:

```
fun {Search.base.one ScriptP}
   {Search.one.depth ScriptP 1 _}
end
```

Suppose that `Script` is a script for which search does not terminate because it keeps on creating choices forever. It could look like the following:

```
proc {Script X}
   ...
   choice {Script X} [] {Script X} end
end
```

If `Search.one.depth` is applied to this particular script by

```
Solutions={Search.one.depth Script 1 KillP}
```

the search engine can be killed by applying `KillP` as follows:

```
{KillP}
```

Note that a script which keeps on computing forever even without search (i.e., because it contains an infinite recursion or loop) can not be killed.

one.depthS

```
{Search.one.depthS +ScriptP +RcdI
                   ?KillP ?Spaces}
```

returns a singleton list containing the first succeeded space for the script +ScriptP (a unary procedure) obtained by depth-first search. If no solution exists, `nil` is returned.

one.depthP

```
{Search.one.depthP +ScriptP +RcdI
                   ?KillP ?Ps}
```

Similar to `Search.one.depthS`, but returns a list of unary procedures as output.

`Search.one.depthP` can be defined using `Search.one.depthS` as follows:

```
fun {Search.one.depthP Script RcdI ?KillP}
   {Map thread
           {Search.one.depthS Script RcdI ?KillP}
        end
        fun {$ SuccSpace}
           proc {$ Root}
              {Space.merge SuccSpace Root}
           end
        end}
end
```

one.bound

```
{Search.one.bound +ScriptP +BoundI +RcdI ?KillP ?Xs}
```

one.boundS

```
{Search.one.boundS +ScriptP +BoundI +RcdI ?KillP ?Spaces
```

one.boundP

```
{Search.one.boundP +ScriptP +BoundI +RcdI ?KillP ?Ps
```

returns a singleton list containing the first solution of the script +ScriptP (a unary procedure) obtained by depth-first search, where the depth of the search tree explored is less than or equal to +BoundI.

If there is no solution in a depth less than or equal to +BoundI, but there might be solutions deeper in the tree, `cut` is returned. In case the entire search tree has a depth less than +BoundI and no solution exists, `nil` is returned.

Otherwise the output is a singleton list containing the solution (`Search.one.bound`), a succeeded space (`Search.one.boundS`), or a procedure (`Search.one.boundP`).

For instance

```
{Search.one.bound proc {$ X}
                     choice fail [] fail end
                  end
                  1 1 _}
```

returns the output `nil`, whereas

```
{Search.one.bound proc {$ X}
                     choice
                        choice fail [] fail end
                     [] choice fail [] fail end
                     end
                  end
                  1 1 _}
```

returns the output `cut`.

`one.iter`

`{Search.one.iter +ScriptP +RcdI ?KillP ?Xs}`

`one.iterS`

`{Search.one.iterS +ScriptP +RcdI ?KillP ?Spaces}`

`one.iterP`

`{Search.one.iterP +ScriptP +RcdI ?KillP ?Ps}`

returns a singleton list containing the first solution of the script `+ScriptP` (a unary procedure) obtained by iterative deepening depth-first search. If no solution exists, `nil` is returned.

Iterative deepening applies `Search.one.bound` to `+ScriptP` with depth-bounds 1, 2, 4, 8, ... until either a solution is found or `Search.one.bound` returns `nil`.

4.2.2 All Solution Search

`all`

`{Search.all +ScriptP +RcdI ?KillP ?Xs}`

`allS`

`{Search.allS +ScriptP +RcdI ?KillP ?Spaces}`

`allP`

`{Search.allP +ScriptP +RcdI ?KillP ?Ps}`

returns the list of all solutions of the script `+ScriptP` (a unary procedure) obtained by depth-first search.

The output is a list of solutions (`Search.all`), a list of succeeded spaces (`Search.allS`), or a list of procedures (`Search.allP`).

4.2.3 Best Solution Search

`best.bab`

`{Search.best.bab +ScriptP +OrderP +RcdI ?KillP ?Xs}`

`best.babS`

`{Search.best.babS +ScriptP +OrderP +RcdI ?KillP ?Spaces}`

best.babP

`{Search.best.babP +ScriptP +OrderP +RcdI ?KillP ?Ps}`

returns a singleton list containing the best solution with respect to the order +OrderP (a binary procedure) of the script +ScriptP (a unary procedure) obtained by branch and bound search. If no solution does exist, `nil` is returned.

The branch and bound strategy works as follows. When a solution is found, all the remaining alternatives are constrained to be *better* with respect to the order +OrderP. The binary procedure +OrderP is applied with its first argument being the previous solution, and its second argument the root variable of a space for one of the remaining alternatives.

best.restart

`{Search.best.restart +ScriptP +OrderP +RcdI ?KillP ?Xs}`

best.restartS

`{Search.best.restartS +ScriptP +OrderP +RcdI ?KillP ?Spaces}`

best.restartP

`{Search.best.restartP +ScriptP +OrderP +RcdI ?KillP ?Ps}`

returns a singleton list containing the best solution with respect to the order +OrderP (a binary procedure) of the script +ScriptP (a unary procedure) obtained by branch and bound search. If no solution does exist, `nil` is returned.

The restart strategy works as follows. When a solution is found, search is restarted for +ScriptP with the additional constraint stating that the solution must be better with respect to the order +OrderP. The binary procedure +OrderP is applied with the previous solution as first argument, and the root variable of the script +ScriptP as its second argument.

4.3 Search.object

The object `Search.object` implements a demand driven search engine which supports recomputation, single, all, and best solution search and different kinds of output in the same way as the search engines in the previous section.

script

```
script(+ScriptP
       +OrderP <= _
       rcd:+RcdI <= 1)
```

Initializes the object for the script +Script (a unary procedure). If the optional argument +OrderP (a binary procedure) is given, the object uses a branch and bound strategy for best solution search.

+RcdI is the recomputation distance (see Section 4.2).

next

`next(?Xs)`

nextS

`nextS(?Spaces)`

nextP

`nextP(?Ps)`

returns a singleton list which contains the next solution. If no further solution exists, `nil` is returned. If the search is stopped by a message `stop`, `stopped` is returned.

The object releases its state immediately.

last

`last(?Xs)`

lastS

`lastS(?Spaces)`

lastP

`lastP(?Ps)`

returns a singleton list which contains the last solution. If no further solution exists, `nil` is returned. If the search is stopped by a message `stop`, `stopped` is returned.

The object releases its state immediately. If the object has been initialized for best solution search, the last solution is the best solution.

stop

`stop`

stops the search engine. The search engine can be restarted by `next`, `nextS`, `nextP`, `last`, `lastS`, and `lastP`.

4.4 Parallel Search Engines

Parallel search engines use multiple networked computers to speed up the exploration of search trees. During exploration of a search tree entire subtrees are delegated to multiple workers. Each worker is powered by a single Oz engine. This means that all worker run in parallel: subtrees are explored in parallel rather than sequentially. Each engine runs on a networked computer, or multiple engines can even run on a single networked computer. The latter makes sense if the computer has more than a single processor and can run the engines in parallel.

When to use? Delegating subtrees for exploration to workers incurs some overhead. But if the number of subtrees is significant, parallel execution can gain over the required overhead. If no subtrees exist (the search tree is just a single path) or the subtrees are small (just a small search tree), parallel search engines do not improve. Branch and bound search for hard problems (like scheduling problems) in particular can take advantage. Currently, you can expect linear speedup for up to six workers (that is, six times faster!) with well suited problems.

What to do? Your scripts do not need rewriting. They must be wrapped into a functor definition.

4.4.1 An Example

Let us take as small constraint problem the fraction problem, which is explained in Section *Example: Fractions, (Finite Domain Constraint Programming in Oz. A Tutorial.)*. However we will choose here a formulation that artificially increases the search tree in that we do not impose a canonical order and leave out redundant constraints.

The script as you can try it from the OPI[1], looks as follows:

29a ⟨**Fractions script** 29a⟩≡
```
proc {Script Root}
   sol(a:A b:B c:C d:D e:E f:F g:G h:H i:I) = Root
   BC = {FD.decl}
   EF = {FD.decl}
   HI = {FD.decl}
in
   Root ::: 1#9
   {FD.distinct Root}
   BC =: 10*B + C
   EF =: 10*E + F
   HI =: 10*H + I
   A*EF*HI + D*BC*HI + G*BC*EF =: BC*EF*HI
   {FD.distribute ff Root}
end
```

It is wrapped into a functor that must export a single feature `script` under which the script (`Fraction` in our case) is available. This is easy, the following does the job:

29b ⟨**Fractions functor** 29b⟩≡
```
functor Fractions
import FD
export Script
define
   ⟨Fractions script 29a⟩
end
```

If you want to learn more about functors, you should consult *"Application Programming"*.

After executing the functor definition in the OPI, we can now start the search engine.

Let us assume that we want to create two processes on the computers with hostname `godzilla` (because it is a double processor machine), and a single process on both `orca` and `grizzly`. We create a parallel search engine that runs on these hosts as follows:

```
E={New Search.parallel init(godzilla:2 orca:1 grizzly:1)}
```

A list of all solutions `Xs` can now be computed as follows:

[1] *"The Oz Programming Interface"*

```
Xs={E all(Fractions $)}
```

Similarly, a single solution Ys can be computed by

```
Xs={E one(Fractions $)}
```

Here, Ys is either a singleton list containg the solution, or nil if no solution does exist. Note that the first solution returned is not necessarily the solution found by the non-parallel search engines first.

Parallel search engines support a (rudimentary) form of tracing. After

```
{E trace(true)}
```

a window appears as that gives graphical information on how many nodes each Oz engine explored. The graphical information is in a very early beta stage and will improve soon.

```
{E trace(false)}
```

switches tracing off again.

Rather than using a functor as an argument for the methods one and all a url can be used that refers to a pickled functor stored under that url.

Search for best solution works similar. Let us consider as a more interesting example the really hard scheduling problem MT10 (for more information on that problem see Section *Solving Hard Scheduling Problems, (Finite Domain Constraint Programming in Oz. A Tutorial.)*). A functor for best solution search must export both script and order. How this is done you can see in the functor definition MT10.oz[2] for the MT10 problem.

Now the list of solutions Zs in strictly increasing order can be computed by

```
Zs={E best('x-oz://doc/system/MT10.ozf' $)}
```

The best solution is the last element of the list Zs. The speed up you can expect is almost a factor of six with six processes started!

Parallel search engines only work properly, if your computing environment is set up such that the facilities for remote module managers work. The requirements are described in Chapter 12.

4.4.2 The Class Search.parallel

The class Search.parallel provides the following methods.

init

```
init(+HostA1:+I1#+ForkA1 ... +HostAn:+In#+ForkAn)
```

[2] MT10.oz

Creates and initializes a new parallel search engine by forking new Oz processes. At host HostA1 the number of newly forked processes is I1 and the fork method ForkA1 is used (see Chapter 12 for a discussion of fork methods), and so on.

For example,

```
E={New Search.parallel init(wallaby:  1#automatic
                            godzilla: 2#ssh
                            grizzly:  1#ssh)}
```

creates a single process at the computer wallaby, two processes at godzilla, and one process at grizzly. The fork method for wallaby is automatically determined, for godzilla and grizzly the method ssh (secure shell) is used.

Equivalently, this can be abbreviated as follows:

```
E={New Search.parallel init(wallaby godzilla:2#ssh grizzly:ssh)}
```

That is, a field with integer feature is assumed to be a host where a single process is to be forked, and the atom automatic for a fork method or the number 1 as number of processes to be forked can be left out.

one

```
one(+FunctorOrUrl ?Xs)
```

Searches a single solution for the script specified by FunctorOrUrl. FunctorOrUrl must be either a functor or a url given as virtual string that refers to a pickled functor. The engine runs the script that must be exported by the field script.

Returns in Xs either nil in case no solution does exists, or a singleton list containing the solution.

Blocks until search terminates.

all

```
all(+FunctorOrUrl ?Xs)
```

Searches all solutions for the script specified by FunctorOrUrl. FunctorOrUrl must be either a functor or a url given as virtual string that refers to a pickled functor. The engine runs the script that must be exported by the field script.

Returns in Xs the list of solutions.

Blocks until search terminates.

best

```
best(+FunctorOrUrl ?Xs)
```

Searches the best solution for the script and order specified by FunctorOrUrl. FunctorOrUrl must be either a functor or a url given as virtual string that refers to a pickled functor. The engine runs the script that must be exported by the field script and uses as order for branch and bound search the fields order.

Returns in Xs either nil in case no solution does exists, or a list containing the solutions in increasing order. That is the last element (if any) is the best solution.

Blocks until search terminates.

stop

```
stop()
```

Stops the current search started by `one`, `all`, or `best`.

Blocks until search has been terminated.

`close`

```
close()
```

Closes the object and terminates all forked Oz processes.

`trace`

```
trace(+B)
```

Switches graphical tracing of search tree delegation on or off, depending on +B.

Method is highly speculative and is subject to change.

5

Finite Domain Constraints: FD

The procedures in this module have the following properties.

Each of their applications creates a new thread except for basic constraints which may block.

Most of the propagators perform interval propagation. Only some do domain propagation (i.e. cut holes into domains).

Equality between variables is exploited, except for some non-linear propagators. For example, `A+A=:B` is equivalent to `2*A=:B` (for notation see sec.infix-ps).

The constraint store is amplified with constraints `X::Spec` and equality between variables, e.g., `X=:Y` is equivalent to `X=Y`.

relation symbols There are generic procedures who take an atomic argument `A` to denote an arithmetic relation $\sim_A$. The possible atoms and the associated relations are summarized below.

A	'=:'	'>:'	'>=:'	'<:'	'=<:'	'\\=:'
$\sim_A$	$=$	$>$	$\geq$	$<$	$\leq$	$\neq$

5.1 Some Facts About Propagators

domain propagation, interval propagation If a propagator is invoked, it tries to narrow the domains of the variables it is posted on. The amount of narrowing of domains depends on the operational semantics of the propagator. There are two main schemes for the operational semantics of a propagator. Domain propagation means that the propagator narrows the domains such that all values are discarded, which are not contained in a solution of the modeled constraint. But due to efficiency reasons, most propagators provide only interval propagation, i.e. only the bounds of domains are narrowed. For some propagators, there is an operational semantics in between both schemes.

A propagator ceases to exist at least if all the variables it is posted on are determined. In the following sections, only exceptions from this rule are mentioned, i.e. if the propagator ceases to exist earlier. For example, {`X =<: Y`} ceases to exist if the current upper bound of `X` is smaller than or equal to the current lower bound of `Y`.

5.2 The Concept of Constructive Disjunction

The operational semantics of some propagators is based on the concept of constructive disjunction which allows to lift common information from different clauses of a disjunctive constraint.

Constructive disjunction is *not* available as program combinator in Oz. Anyway, we use it in Oz program fragments (by the keyword `condis`) to describe the operational semantics of certain propagators. For example such propagators are `FD.tasksOverlap` (page 47) and `FD.disjoint` (page 47).

Constructive disjunction adopts the operational semantics of the nondistributable disjunction of Oz (**`or ... end`**) concerning entailment and failure of clauses. Furthermore, it extends the semantics as follows: Assume a disjunction with n clauses and let S be the constraint store of the computation space in which it resides. Let $S_1,\ldots,S_n$ denote the local stores of the n clauses. Then the strongest constraint C consisting of basic constraints $X \in D$ with $S_i \models C$ for $1 \leq i \leq n$ is lifted and added to S.

As an example consider the store `X, Y`$\in \{0,\ldots,10\}$ and

```
condis X + 9 =<: Y
[] Y + 9 =<: X
end
```

Constructive disjunction narrows the domains of `X` and `Y` to $\{0,1,9,10\}$.

5.3 Finite Domains

`inf`

```
FD.inf
```

is a constant integer. Its concrete value is implementation dependent. In Mozart `FD.inf` is 0.

`sup`

```
FD.sup
```

is a constant integer. Its concrete value is implementation dependent. In Mozart `FD.sup` is 134 217 726.

`is`

```
{FD.is *D ?B}
```

tests whether `D` is an integer between 0 and `FD.sup`.

5.4 Telling Domains

`::`

```
?D::+Spec
{FD.int +Spec ?D}
```

tells the constraint store that `D` is an integer in `Spec`.

`:::`

`Dv:::+Spec`
`{FD.dom +Spec ?Dv}`

tells the constraint store that `Dv` is a vector of integers in `Spec`. Waits until `Dv` is constrained to a vector.

`list`

`{FD.list +I +Spec ?Ds}`

tells the constraint store that `Ds` is a list of integers in `Spec` of length `I`.

`tuple`

`{FD.tuple +L +I +Spec ?Dt}`

tells the constraint store that `Dt` is a tuple of integers in `Spec` of width `I` and label `L`.

`record`

`{FD.record +L +Ls +Spec ?Dr}`

tells the constraint store that `Dr` is a record of integers in `Spec` with features `Ls` and label `L`.

`decl`

`{FD.decl ?D}`

Abbreviates `{FD.int 0#FD.sup D}`.

5.5 Reflection

`reflect.min`

`{FD.reflect.min *D1 ?D2}`

returns the current lower bound of `D1`.

`reflect.max`

`{FD.reflect.max *D1 ?D2}`

returns the current upper bound of `D1`.

`reflect.mid`

`{FD.reflect.mid *D1 ?D2}`

returns the integer which is closest to the middle of the current domain (the arithmetical means of the lower and upper bound of `D1`). In case of ties, the smaller element is selected.

`reflect.nextLarger`

`{FD.reflect.nextLarger *D1 +D2 ?D3}`

returns the smallest integer in the domain of `D1` which is larger than `D2`.

`reflect.nextSmaller`

`{FD.reflect.nextSmaller *D1 +D2 ?D3}`

returns the largest integer in the domain of `D1` which is smaller than `D2`.

`reflect.size`

`{FD.reflect.size *D1 ?D2}`

returns the size of the current domain of `D1`.

`reflect.domList`

`{FD.reflect.domList *D ?Ds}`

returns the current domain of `D` as an ordered list of integers.

`reflect.dom`

`{FD.reflect.dom *D ?Spec}`

returns the current domain of `D` as a domain specification.

`reflect.nbSusps`

`{FD.reflect.nbSusps *D ?I}`

returns the current number of suspensions on `D`.

5.6 Watching Domains

`watch.min`

`{FD.watch.min *D1 +D2 ?B}`

Returns `true` when $\mathtt{D1} \in \{\mathtt{D2}+1,\ldots,\mathtt{FD.sup}\}$ and `false` when $\mathtt{D1} \in \{0,\ldots,\mathtt{D2}\}$ is entailed by the constraint store.

`watch.max`

`{FD.watch.max *D1 +D2 ?B}`

Returns `true` when $\mathtt{D1} \in \{0,\ldots,\mathtt{D2}-1\}$ and `false` when $\mathtt{D1} \in \{D2,\ldots,\mathtt{FD.sup}\}$ is entailed by the constraint store.

`watch.size`

`{FD.watch.size *D1 +D2 ?B}`

Returns `true` when the size of the domain of `D1` becomes smaller than `D2`.

5.7 Generic Propagators

The generic propagators `FD.sum`, `FD.sumC` and `FD.sumCN` do interval propagation. The propagators `FD.sumAC` and `FD.sumACN` do interval propagation but may also cut holes into domains. For example,

```
{FD.dom 0#10 [X Y]}
{FD.sumAC [1 ~1] [X Y] '>:' 8}
```

will reduce the domains of `X` and `Y` to $\{0,1,9,10\}$. Except for propagators `FD.sumCN` and `FD.sumACN`, equality is exploited, e.g. `{FD.sumC [2 3] [A A] '=:' 10}` is equivalent to `{FD.sumC [5] [A] '=:' 10}`.

$\underline{x}$,$\overline{x}$ Let S denote the current constraint store and x a finite domain integer.$\underline{x}$ denotes the largest integer such that $S \models x \geq \underline{x}$ holds. Analogously, $\overline{x}$ denotes the smallest integer such that $S \models x \leq \overline{x}$ holds.

$\lfloor n \rfloor$, $\lceil n \rceil$ Let n denote a real number. $\lfloor n \rfloor$ denotes the largest integer which is equal or smaller than n. Analogously, $\lceil n \rceil$ denotes the smallest integer which is equal or larger than n.

sum

```
{FD.sum *Dv +A *D}
```

creates a propagator for

$1 * \mathtt{D}_1 + \ldots + 1 * \mathtt{D}_n + (-1) * \mathtt{D} \sim_{\mathtt{A}} 0$

For the operational semantics see `FD.sumC`. For the relation symbol `'\\=:'`, the propagator waits until at most one non-determined variable is left. Then the appropriate value is discarded from the variable's domain. For the other relations, the propagator does interval propagation.

sumC

```
{FD.sumC +Iv *Dv +A *D}
```

creates a propagator for the scalar product of the vectors `Iv` and `Dv`:

$\mathtt{I}_1 * \mathtt{D}_1 + \ldots + \mathtt{I}_n * \mathtt{D}_n + (-1) * \mathtt{D} \sim_{\mathtt{A}} 0$

Let $\mathtt{D}_{n+1}$ be `D` and $\mathtt{I}_{n+1}$ be -1. Then, the operational semantics is defined as follows. For each product $\mathtt{I}_k * \mathtt{D}_k$, an isolation (projection) is computed:

$$\mathtt{I}_k * \mathtt{D}_k \sim_{\mathtt{A}} \underbrace{- \sum_{i=1, i \neq k}^{n+1} \mathtt{I}_i * \mathtt{D}_i}_{RHS_k}.$$

For the right hand side RHS_k, the upper $\overline{RHS_k}$ and lower limit $\underline{RHS_k}$ are defined as follows.

$$\begin{aligned} \overline{RHS_k} &= - \sum_{i=1, i \neq k, \mathtt{I}_i > 0}^{n+1} \mathtt{I}_i * \underline{\mathtt{D}_i} - \sum_{i=1, i \neq k, \mathtt{I}_i < 0}^{n+1} \mathtt{I}_i * \overline{\mathtt{D}_i} \\ \underline{RHS_k} &= - \sum_{i=1, i \neq k, \mathtt{I}_i > 0}^{n+1} \mathtt{I}_i * \overline{\mathtt{D}_i} - \sum_{i=1, i \neq k, \mathtt{I}_i < 0}^{n+1} \mathtt{I}_i * \underline{\mathtt{D}_i} \end{aligned}$$

These values are used to narrow the domain of $\mathtt{D}_k$ until a fixed point is reached. We describe the propagation for the different possible values of `A`.

'=<:'

Narrowing is done according to the following inequalities.

$$\mathtt{D}_k \le \left\lfloor \frac{\overline{RHS_k}}{\mathtt{I}_k} \right\rfloor \quad \text{if } \mathtt{I}_k > 0$$

$$\mathtt{D}_k \ge \left\lceil \frac{\overline{RHS_k}}{\mathtt{I}_k} \right\rceil \quad \text{if } \mathtt{I}_k < 0$$

Here $x \le n$ denotes the basic constraint $x \in \{0, \ldots, n\}$ and $x \ge n$ denotes the basic constraint $x \in \{n, \ldots, \mathtt{FD.sup}\}$.

The propagator ceases to exist, if the following condition holds.

$\sum_{i=1, \mathtt{I}_i > 0}^{n+1} \mathtt{I}_i * \overline{\mathtt{D}_i} + \sum_{i=1, \mathtt{I}_i < 0}^{n+1} \mathtt{I}_i * \underline{\mathtt{D}_i} \le 0$

As an example consider

```
X - Y =<: Z - V
```

We have the following narrowing.

$X \le \overline{Z} - \underline{V} + \overline{Y} \qquad Y \ge \underline{X} - \overline{Z} + \overline{V} \qquad Z \ge \underline{X} - \overline{Y} + \underline{V} \qquad V \le \overline{Z} - \underline{X} + \overline{Y}$

The propagator ceases to exist if $\overline{X} - \underline{Y} \le \underline{Z} - \overline{V}$ holds.

`'>=:'`

This case can be reduced to `=<:` due to the observation that

$I_1 * D_1 + \ldots + I_n * D_n + (-1) * D \ \ge \ 0$

is equivalent to

$(-I_1) * D_1 + \ldots + (-I_n) * D_n + 1 * D \ \le \ 0$

Alternatively, $\underline{RHS_k}$ can be used for the definition.

`'<:'`

Analogous to `'=<:'`

`'>:'`

Analogous to `'>=:'`

`'=:'`

In this case, the operational semantics is defined by conjunction of the formulas given for `=<:` and `>=:`. Furthermore, coreferences are realized in that, e.g. the propagator `3*X=:3*Y` tells the basic constraint `X=Y`.

`'\\=:'`

In this case, the propagator waits until at most one non-determined variable is left, say D_k. Then, RHS_k denotes a unique integer value which is discarded from the domain of D_k.

Additional propagation is achieved through the realization of coreferences, i.e. equality between variables. If the store S entails (without loss of generality) $D_1 = D_2$, the generic propagator evolves into:

$(I_1 + I_2) * D_2 + \ldots + I_n * D_n + (-1) * D \ \sim_A \ 0$

sumCN

```
{FD.sumCN +Iv *Dvv +A *D}
```

creates a propagator for

$I_1 * D_{11} * \ldots * D_{1m_1} + \ldots + I_n * D_{n1} * \ldots * D_{nm_n} + (-1) * D \ \sim_A \ 0$

Let $D_{(n+1)1}$ be D, I_{n+1} be -1, and m_{n+1} be 1. Then, the operational semantics is defined as follows. For $k, 1 \le k \le n+1$, an isolation (projection) is computed:

$$I_k * D_{k1} * \ldots * D_{km_k} \sim_A \underbrace{- \sum_{i=1, i \ne k}^{n+1} I_i * \prod_{j=1}^{m_i} D_{ij}}_{RHS_k}$$

For the right hand side RHS_k, the upper $\overline{RHS_k}$ and lower limit $\underline{RHS_k}$ are defined as follows.

$$\begin{aligned} \overline{RHS_k} &= - \sum_{i=1, i \ne k, I_i > 0}^{n+1} I_i * \prod_{j=1}^{m_i} \underline{D}_{ij} - \sum_{i=1, i \ne k, I_i < 0}^{n+1} I_i * \prod_{j=1}^{m_i} \overline{D}_{ij} \\ \underline{RHS_k} &= - \sum_{i=1, i \ne k, I_i > 0}^{n+1} I_i * \prod_{j=1}^{m_i} \overline{D}_{ij} - \sum_{i=1, i \ne k, I_i < 0}^{n+1} I_i * \prod_{j=1}^{m_i} \underline{D}_{ij} \end{aligned}$$

These values are used to narrow the domain of $\mathtt{D}_{kl}, 1 \leq l \leq m_k$, until a fixed point is reached. We describe the propagation for the different possible values of `A`.

`'=<:'`

The narrowing is done according to the following inequalities.

$$\mathtt{D}_{kl} \leq \left\lfloor \frac{\overline{RHS_k}}{\mathtt{I}_k * \prod_{j=1,j\neq l}^{m_k} \underline{\mathtt{D}}_{kj}} \right\rfloor \qquad \text{if } \mathtt{I}_k > 0$$

$$\mathtt{D}_{kl} \geq \left\lceil \frac{\overline{RHS_k}}{\mathtt{I}_k * \prod_{j=0,j\neq l}^{m_k} \overline{\mathtt{D}}_{kj}} \right\rceil \qquad \text{if } \mathtt{I}_k < 0$$

Here $x \leq n$ denotes the basic constraint $x \in \{0, \ldots, n\}$ and $x \geq n$ denotes the basic constraint $x \in \{n, \ldots, \mathtt{FD.sup}\}$.
The propagator ceases to exist, if the following condition holds.
$\sum_{i=1,\mathtt{I}_i>0}^{n+1} \mathtt{I}_i * \prod_{j=1}^{m_i} \overline{\mathtt{D}}_{ij} + \sum_{i=1,\mathtt{I}_i<0}^{n+1} \mathtt{I}_i * \prod_{j=1}^{m_i} \underline{\mathtt{D}}_i \leq 0$
As an example consider

```
3*X*Y - Z =<: A
```

We have the following formulas.

$$\mathtt{X} \leq \left\lfloor \frac{\overline{\mathtt{A}} + \overline{\mathtt{Z}}}{3 * \underline{\mathtt{Y}}} \right\rfloor \qquad \mathtt{Y} \leq \left\lfloor \frac{\overline{\mathtt{A}} + \overline{\mathtt{Z}}}{3 * \underline{\mathtt{X}}} \right\rfloor \qquad \mathtt{Z} \geq \underline{\mathtt{X}} * \overline{\mathtt{Y}} - \overline{\mathtt{A}} \qquad \mathtt{A} \geq \underline{\mathtt{X}} * \overline{\mathtt{Y}} - \overline{\mathtt{Z}}$$

The propagator ceases to exist if $3 * \overline{\mathtt{X}} * \overline{\mathtt{Y}} - \underline{\mathtt{Z}} \leq \underline{\mathtt{A}}$ holds.

`'>=:'`

This case can be reduced to `'=<:'` due to the observation that
$\mathtt{I}_1 * \mathtt{D}_{11} * \ldots * \mathtt{D}_{1k_1} + \ldots + \mathtt{I}_n * \mathtt{D}_{n1} * \ldots * \mathtt{D}_{nk_n} + (-1) * \mathtt{D}_{(n+1)1} \ \leq \ 0$
is equivalent to
$(-\mathtt{I}_1) * \mathtt{D}_{11} * \ldots * \mathtt{D}_{1k_1} + \ldots + (-\mathtt{I}_n) * \mathtt{D}_{n1} * \ldots * \mathtt{D}_{nk_n} + 1 * \mathtt{D}_{(n+1)1} \ \geq \ 0$
Alternatively, $\underline{RHS_k}$ can be used for the definition.

`'<:'`

Analogous to `'=<:'`

`'>:'`

Analogous to `'>=:'`

`'=:'`

In this case, the operational semantics is defined by conjunction of the formulas given for `'=<:'` and `'>=:'`.

`'\\=:'`

In this case, the propagator waits until at most one non-determined variable is left, say D_{kl}. Then, RHS_k denotes a unique integer, and if
$\frac{RHS_k}{\mathtt{I}_k * \prod_{j=1,j\neq l}^{m_k} \mathtt{D}_{kj}}$
denotes an integer value, this value is discarded from the domain of $\mathtt{D}_{kl}$.

Coreferences are not exploited for nonlinear generic constraints. The only exception is the expression

```
X * X =: Y
```

which has the same operational semantics as `{FD.times X X Y}` (but note that the occurring variables are not automatically constrained to finite domain integers).

sumAC

```
{FD.sumAC +Iv *Dv +A *D}
```

creates a propagator for the absolute value of the scalar product of the vectors `Iv` and `Dv`:

$|\mathtt{Iv} * \mathtt{Dv}| = |\mathtt{I}_1 * \mathtt{D}_1 + \ldots + \mathtt{I}_n * \mathtt{D}_n| \sim_{\mathtt{A}} \mathtt{D}$

The operational semantics is as follows. If A is `'<:'`, `'=<:'` or `'\\=:'`, the following definition holds.

$\mathtt{Iv} * \mathtt{Dv} \sim_{\mathtt{A}} D \wedge (-\mathtt{Iv}) * \mathtt{Dv} \sim_{\mathtt{A}} D$

If A is `'>:'`, `'>=:'` or `'=:'`, the following definition holds.

$\mathtt{Iv} * \mathtt{Dv} \sim_{\mathtt{A}} D \vee (-\mathtt{Iv}) * \mathtt{Dv} \sim_{\mathtt{A}} D$

where the disjunction is realized by constructive disjunction.

sumACN

```
{FD.sumACN +Iv *Dvv +A *D}
```

creates a propagator for

$|\mathtt{I}_1 * \mathtt{D}_{11} * \ldots * \mathtt{D}_{1k_1} + \ldots + \mathtt{I}_n * \mathtt{D}_{n1} * \ldots * \mathtt{D}_{nk_n}| \sim_{\mathtt{A}} \mathtt{D}$

The operational semantics is defined analogously to `FD.sumAC`.

sumD

```
{FD.sumD *Dv +A *D}
```

creates a propagator analogous to `FD.sum` but performs *domain-consistent* propagation. Note that only equality (A is `'=:'`) and disequality (A is `'\\=:'`) are supported, as for inequalities domain and bounds propagation are equivalent.

sumCD

```
{FD.sumCD +Iv *Dv +A *D}
```

creates a propagator analogous to `FD.sumC` but performs *domain-consistent* propagation. Note that only equality (A is `'=:'`) and disequality (A is `'\\=:'`) are supported, as for inequalities domain and bounds propagation are equivalent.

5.8 Symbolic Propagators

The following propagators do domain propagation or amplify the store by constraints `X::Spec`, where `Spec` may also contain holes.

distinct

```
{FD.distinct *Dv}
```

All elements in `Dv` are pairwise distinct. If one element becomes determined, the remaining elements are constrained to be different from it. If two variables become equal, the propagator fails, e.g. `{FD.distinct [A A B]}` will fail even if `A` is not determined.

distinctB

```
{FD.distinctB *Dv}
```

All elements in `Dv` are pairwise distinct. Uses bounds propagation, but does not use value propagation as `FD.distinct`. Also fails, if two variables are equal. Currently uses the quadratic algorithm for propagation by Puget described in [7].

`distinctD`

```
{FD.distinctD *Dv}
```

All elements in `Dv` are pairwise distinct. Uses full domain propagation. Also fails, if two variables are equal. Is based on Régin's algorithm [8].

`distinctOffset`

```
{FD.distinctOffset *Dv +Iv}
```

All sums $\mathtt{D}_i + \mathtt{I}_i$ are pairwise distinct, i.e. for all $i \neq j$ holds $\mathtt{D}_i + I_i \neq \mathtt{D}_j + \mathtt{I}_j$. If one $\mathtt{D}_i$ becomes determined, the remaining elements $\mathtt{D}_j$ are constrained to be different from $\mathtt{D}_i + \mathtt{I}_i - \mathtt{I}_j$.

`distinct2`

```
{FD.distinct2 *Dv1 +Iv1 *Dv2 +Iv2}
```

Assume that all arguments are tuples of width n. Then the propagator's operational semantics is defined as follows.

```
or Dv1.i + IV1.i =<: Dv1.j
[] Dv1.j + IV1.j =<: Dv1.i
[] Dv2.i + IV2.i =<: Dv2.j
[] Dv2.j + IV2.j =<: Dv2.i
end
```

This propagator may be used to express that a number of rectangles must not overlap in the two-dimensional space. In this case `Dv1` and `Dv2` may denote the x-coordinates and y-coordinates of the lower left corner of the rectangles, respectively. `Iv1` and `Iv2` may denote the widths and heights of the rectangles, respectively.

`atMost`

```
{FD.atMost *D *Dv +I}
```

`atLeast`

```
{FD.atLeast *D *Dv +I}
```

`exactly`

```
{FD.exactly *D *Dv +I}
```

At most, at least, exactly `D` elements of `Dv` are equal to `I`. The operational semantics is defined as follows. Let `VFoldL` be either `FoldL` or `Record.foldL` depending on the type of `Dv` and

```
S = {VFoldL Dv fun{$ In D1} {FD.plus In D1=:I} end 0}
```

The propagator `FD.atMost`, `FD.atLeast` and `FD.exactly` are defined by `D>=:S`, `D=<:S` and `D=:S`, respectively.

`element`

```
{FD.element *D1 +Iv *D2}
```

The D1-th element of Iv is D2.

It propagates as follows. For each integer *i* in the domain of D1, the *i*-th element of Iv is in the domain of D2; and no other values. For each value *j* in the domain of D2, all positions where *j* occurs in Is are in the domain of D1; and no other values. For example,

```
{FD.int [1 3] X} {FD.element X [5 6 7 8] Y}
```

will constrain Y to $\{5,7\}$. D1 is constrained to be greater than 0.

5.9 0/1 Propagators

Using the mapping from 0 and 1 to the truth values false and true, respectively, logical connectives between finite domain integers are defined. If at most one argument is a free variable, it will be constrained to a finite domain integer in $\{0,1\}$. Such a finite domain integer is also called a 0/1-integer. The propagators exploit equality and may also post equality between variables.

The operational semantics is detailed only for FD.conj. For the remaining propagators, the operational semantics is defined accordingly, exploiting as much information as possible (including coreferences).

conj

```
{FD.conj $D1 $D2 $D3}
```

D3 is the conjunction of D1 and D2. The operational semantics can be described by the following code

```
[D1 D2 D3] ::: 0#1
cond D1=0  then D3=0
[]   D1=1  then D2=D3
[]   D2=0  then D3=0
[]   D2=1  then D1=D3
[]   D3=1  then D1=1 D2=1
[]   D1=D2 then D1=D3
end
```

disj

```
{FD.disj $D1 $D2 $D3}
```

D3 is the disjunction of D1 and D2.

exor

```
{FD.exor $D1 $D2 $D3}
```

D3 is the exclusive disjunction of D1 and D2.

nega

```
{FD.nega $D1 $D2}
```

D2 is the negation of D1.

impl

```
{FD.impl $D1 $D2 $D3}
```

D3 is the implication of D2 by D1 (‘D1 $\longrightarrow$ D2’).

equi

```
{FD.equi $D1 $D2 $D3}
```

D3 is the equivalence of D1 by D2 (‘D1 $\leftrightarrow$ D2’).

5.10 Reified Constraints

Reified constraints reflect the validity of a constraint *C* into a 0/1-valued finite domain integer. The propagator realizing a reified constraint is called the reification propagator. The reification propagators wait in the same way as their non-reified counterparts. All reification propagators constrain their last argument to a 0/1-valued finite domain integer.

Let *C* be a constraint and *P* the corresponding propagator. Reifying *C* into a 0/1-valued variable `D` is defined by

$(C \leftrightarrow \mathtt{D} = 1) \wedge \mathtt{D} \in \{0,1\}$.

This is implemented by

```
D::0#1
or P D=1
[] P^N D=0
end
```

Here, P^N denotes the negation of *P* (i.e. a propagator for the negation of the denotational semantics of *P*).

If *P* is one of `{FD.reified.int Spec X}` and `{FD.reified.dom Spec Xv}`, then P^N denotes `{FD.reified.int ComplSpec X}` and `{FD.reified.dom ComplSpec Xv}`, respectively (where `ComplSpec = compl(Spec)` if `Spec` is a simple domain specification, and `ComplSpec = SSpec` if `Spec = compl(SSpec)`).

For the propagators *P* wich are parameterized by a relation symbol *A*, the symbol of the negated relation occurs in P^N. For instance, if *P* is `{FD.sum Ds '<:' X Y}`, then P^N is `{FD.sum Ds '>=:' X Y}`.

reified.int

```
{FD.reified.int +Spec *D1 D2}
```

reifies `{FD.int Spec D1}` into D2.

reified.dom

```
{FD.reified.dom +Spec Dv D}
```

reifies `{FD.dom Spec Dv}` into D.

reified.sum

```
{FD.reified.sum *Dv +A *D1 D2}
```

reifies `{FD.sum Dv A D1}` into D2.

reified.sumC

`{FD.reified.sumC +Iv *Dv +A *D1 D2}`

reifies `{FD.sumC Iv Dv A D1}` into `D2`.

`reified.sumCN`

`{FD.reified.sumCN +Iv *Dvv +A *D1 D2}`

reifies `{FD.sumCN Iv Dvv A D1}` into `D2`.

`reified.sumAC`

`{FD.reified.sumAC +Iv *Dv +A *D1 D2}`

reifies `{FD.sumAC Iv Dv A D1}` into `D2`.

`reified.sumACN`

`{FD.reified.sumACN +Iv *Dvv +A *D1 D2}`

reifies `{FD.sumACN Iv Dvv A D1}` into `D2`.

`reified.distance`

`{FD.reified.distance *D1 *D2 +A *D3 D4}`

reifies `{FD.distance D1 D2 A D3}` into `D4`.

`reified.card`

`{FD.reified.card *D1 *Dv *D2 D3}`

Dv is a vector of Boolean variables. `FD.reified.card` creates a propagator for

$((\mathtt{D1} \leq \mathrm{D}_1 + \ldots + \mathrm{D}_n \leq \mathtt{D2}) \leftrightarrow (\mathtt{D3} = 1)) \wedge \mathtt{D3} \in \{0,1\}$.

which reifies into `D3` the conjunction

```
D1 =<: D1 + ... + Dn
D1 + ... + Dn =<: D2
```

More specifically, its operational semantics is defined through

```
D3 :: 0#1
or D1 =<: D1 + ... + Dn
   D1 + ... + Dn =<: D2
   D3 = 1
[] or D1 >: D1 + ... + Dn
   [] D1 + ... + Dn >: D2
   end
   D3 = 0
end
```

5.11 Miscellaneous Propagators

`plus`

`{FD.plus $D1 $D2 $D3}`

`D3` is the sum of `D1` and `D2`. The propagator constrains its arguments as `D1+D2=:D3`.

`plusD`

`{FD.plusD $D1 $D2 $D3}`

D3 is the sum of D1 and D2. The propagator constrains its arguments as D1+D2=:D3.

Does domain propagation, which can be very expensive.

minus

```
{FD.minus $D1 $D2 $D3}
```

D3 is the difference between D1 and D2. The propagator constrains its arguments as D1-D2=:D3.

minusD

```
{FD.minusD $D1 $D2 $D3}
```

D3 is the difference between D1 and D2. The propagator constrains its arguments as D1-D2=:D3.

Does domain propagation, which can be very expensive.

times

```
{FD.times $D1 $D2 $D3}
```

D3 is the product of D1 and D2. Coreferences are exploited. If the store entails D1 = D3, the propagator ceases to exist and the constraint D2=1 is imposed. If the store entails D2 = D3, the propagator ceases to exist and the constraint D1=1 is imposed. If the store entails D1 = D2, the propagator ceases to exist and a propagator is imposed instead, which constrains the variables D1 and D2 as follows.

$\underline{D1}^2 \le D3 \le \overline{D1}^2 \qquad \lceil\sqrt{\underline{D3}}\rceil \le D1 \le \lfloor\sqrt{\overline{D3}}\rfloor$

For notation see Section 5.7n.

timesD

```
{FD.timesD $D1 $D2 $D3}
```

D3 is the product of D1 and D2.

Does domain propagation, which can be very expensive.

power

```
{FD.power $D1 +I $D2}
```

$D2 is the result of D1 raised to the power of I, i.e. $D1^I = D2$. The propagator constrains the variables D1 and D2 as follows.

$\underline{D1}^I \le D2 \le \overline{D1}^I \qquad \lceil\sqrt[D2]{\underline{D1}}\rceil \le D2 \le \lfloor\sqrt[D2]{\overline{D1}}\rfloor$

For notation see Section 5.7.

divI

```
{FD.divI $D1 +I $D2}
```

D2 is the result of the integer division of D1 by I.

A domain bound is discarded from the domain of one variable, if there is no value between the lower and upper bound of the domain of the other variable, such that the constraint holds. Additionally, if $D1 = D2$, the propagator is replaced by I=1.

modI

```
{FD.modI $D1 +I $D2}
```

D2 is the result of D1 modulus I.

A domain bound is discarded from the domain of one variable, if there is no value between the lower and upper bound of the domain of the other variable, such that the constraint holds. Additionally, if D1 = D2, the propagator is replaced by D1<:I. If the current upper bound of D1 is less than I, the propagator is replaced by D1=D2.

divD

```
{FD.divD $D1 +I $D2}
```

D2 is the result of the integer division of D1 by I.

Does domain propagation, which can be very expensive.

modD

```
{FD.modD $D1 +I $D2}
```

D2 is the result of D1 modulus I.

Does domain propagation, which can be very expensive.

max

```
{FD.max $D1 $D2 $D3}
```

D3 is the maximum of D1 and D2.

Its operational semantics is defined through

```
D3>=:D1   D3>=:D2
condis D3=<:D1
[] D3=<:D2
end
if D1=D2 then D3=D1
else skip
end
```

min

```
{FD.min $D1 $D2 $D3}
```

D3 is the minimum of D1 and D2. Its operational semantics is defined through

```
D3=<:D1   D3=<:D2
condis D3>=:D1
[] D3>=:D2
end
if D1=D2 then D3=D1
else skip
end
```

distance

```
{FD.distance *D1 *D2 +A *D3}
```

creates a propagator for $|\texttt{D1} - \texttt{D2}| \sim_{\texttt{A}} \texttt{D3}$. May cut holes into domains. For example,

```
{FD.dom 0#10 [X Y]}
{FD.distance X Y '>:' 8}
```

will reduce the domains of `X` and `Y` to $\{0,1,9,10\}$.

The propagator is equivalent to `{FD.sumAC [1 ~1] [D1 D2] A D3}` but is more efficient.

`less`

```
{FD.less *D1 *D2}
```

Equivalent to `D1<:D2`.

`lesseq`

```
{FD.lesseq *D1 *D2}
```

Equivalent to `D1 =<: D2`.

`greater`

```
{FD.greater *D1 *D2}
```

Equivalent to `D1>:D2`.

`greatereq`

```
{FD.greatereq *D1 *D2}
```

Equivalent to `D1>=:D2`.

`disjoint`

```
{FD.disjoint *D1 +I1 *D2 +I2}
```

creates a propagator for $\mathtt{D1}+\mathtt{I1}\leq\mathtt{D2}\ \vee\ \mathtt{D2}+\mathtt{I2}\leq\mathtt{D1}$. May cut holes into domains. For example,

```
{FD.dom 0#10 [X Y]}
{FD.disjoint X 9 Y 9}
```

will reduce the domains of `X` and `Y` to $\{0,1,9,10\}$.

Its operational semantics is defined through

```
condis D1 + I1 =<: D2
[] D2 + I2 =<: D1
end
```

`disjointC`

```
{FD.disjointC *D1 +I1 *D2 +I2 D3}
```

creates a propagator for

$((\mathtt{D1}+\mathtt{I1}\leq\mathtt{D2}\wedge\mathtt{D3}=0)\ \vee\ (\mathtt{D2}+\mathtt{I2}\leq\mathtt{D1}\wedge\mathtt{D3}=1))\ \wedge\ (\mathtt{D3}\in\{0,1\})$.

Its operational semantics is defined through

```
condis D1 + I1 =<: D2
   D3 =: 0
[] D2 + I2 =<: D1
   D3 =: 1
end
```

`tasksOverlap`

```
{FD.tasksOverlap *D1 +I1 *D2 +I2 D3}
```

creates a propagator for

$((\mathtt{D1}+\mathtt{I1}>\mathtt{D2}\ \wedge\ \mathtt{D2}+\mathtt{I2}>\mathtt{D1}\ \wedge\ \mathtt{D3}=1)\ \vee\ (\mathtt{D1}+\mathtt{I1}\leq\mathtt{D2}\ \wedge\ \mathtt{D3}=0)\ \vee\ (\mathtt{D2}+\mathtt{I2}\leq\mathtt{D1}\ \wedge\ \mathtt{D3}=0))\ \wedge\ (\mathtt{D3}\in\{0,1\})$.

Its operational semantics is defined through

```
condis
   D1 + I1 >: D2
   D2 + I2 >: D1
   D3 =: 1
[]
   D1 + I1 =<: D2
   D3 =: 0
[]
   D2 + I2 =<: D1
   D3 =: 0
end
```

Note that the disjunction is constructive (page 34). Informally, in case D3 is 0 the propagator behaves like FD.disjoint, i.e., in context of task scheduling two tasks must not overlap. Otherwise, if D3 is 1, the two tasks must overlap. This propagator is used in applications which shall be able to deal with overlapping tasks.

5.12 Distribution

In this section it is shown how Oz supports distribution with constraints. The following procedure creates binary choice-points for variables. The choice is delayed until propagation has reached a fixed point. Assume Dv to be a vector of finite domain integers. The distribution differs in the order of the choice-points and in the constraint with which is distributed. Essentially, it works as follows

- Select an element D of Dv which is not determined.
- Select a value or a domain specification Spec in the current domain of D.
- Create a choice point for X::Spec and X::compl(Spec).
- If not all elements of Dv are determined, go to step 1.

The order of Dv is preserved.

distribute

{FD.distribute +Dist +Xv}

The vector Xv is distributed according to the specification Dist. Dist may be either the atom naive, ff (for first-fail), split or a record with label generic:

- naive: Xv must be a vector of finite domain integers. Considers only non-determined elements of Xv. Chooses the leftmost variable X in Xv. Creates a choice point for X=L and X\=:L, where L is the lower bound of the domain of X.

- `ff`: `Xv` must be a vector of finite domain integers. Considers only non-determined elements of `Xv`. Chooses the leftmost variable `X` in `Xv`, whose domain size is minimal. Creates a choice point for `X=L` and `X\=:L`, where `L` is the lower bound of the domain of `X`.
- `split`: `Xv` must be a vector of finite domain integers. Considers only non-determined elements of `Xv`. Chooses the leftmost variable `X` in `Xv`, whose domain size is minimal. Creates a choice point for `X=<:M` and `X>:M`, where `M` is the middle of the domain of `X` (see `FD.reflect.mid`).
-
  ```
  generic(order:     +Order   <= size
          filter:    +Filter  <= undet
          select:    +Select  <= id
          value:     +Value   <= min
          procedure: +Proc    <= proc {$} skip end)
  ```
  Considers only those elements in `Xv`, for which `Filter` is true. Chooses the leftmost element, which is minimal with respect to `Order` and selects the corresponding variable `D` by `Select`. Creates a choice point for `D::Spec` and `D::compl(Spec)`, where `Spec` is selected by `Value`.

  The values under the respective features must be as follows:
  - `Order`:
    - Binary boolean function `P`: Selects the leftmost element in `Xv` which is minimal with respect to the order relation `P`.
    - `naive`: Selects the leftmost variable.
    - `size`: Selects the leftmost variable, whose domain is minimal.
    - `min`: Selects the leftmost variable, whose lower bound is minimal.
    - `max`: Selects the leftmost variable, whose upper bound is maximal.
    - `nbSusps`: Selects the variable with the largest number of suspensions. If several variables suspend on the maximal number of constraints, the leftmost variable whose domain is minimal is selected.
  - `Filter`:
    - Unary boolean function `P`: Considers only the elements `X` in `Xv`, for which `{P X}` yields `true`.
    - `undet`: Considers only undetermined variables.
  - `Select`:
    - Unary function `P`: Selects the variable to enumerate from the selected element by `Order` and `Filter`.
    - `id`: The variable to enumerate is the selected element.
  - `Value`:
    - Binary procedure `P`: Takes a variable as first argument, and binds its second argument to a domain descriptor `D` to serve as the restriction on said variable to be used in a binary distribution step (`D` in one branch, `compl(D)` in the other).
    - `min`: Selects the lower bound of the domain.
    - `max`: Selects the upper bound of the domain.
    - `mid`: Selects the element, which is closest to the middle of the domain (the arithmetical means between the lower and upper bound of the domain). In case of ties, the smaller element is selected.

  * `splitMin`: Selects the interval from the lower bound to the middle of the domain (see `mid`).
  * `splitMax`: Selects the interval from the element following the middle to the upper bound of the domain (see `mid`).
- `Proc`: Is applied when stability is reached. Since this application may cause instability, distribution is continued when stability is reached again.

Note, that in case `Det` is `det` or in case `Order` is `size`, `lower`, `upper`, or `nbSusps`, the elements of `Xv` must be finite domain integers.

For example, `{FD.distribute ff Dv}` can be expressed as

```
{FD.distribute generic Dv},
```

`{FD.distribute split Dv}` as

```
{FD.distribute generic(value: splitMin) Dv},
```

and `{FD.distribute naive Dv}` as

```
{FD.distribute generic(order: naive) Dv}
```

The naive distribution can also be defined as follows using the `value` feature.

```
{FD.distribute
generic(value: fun {$ D}
                  {FD.reflect.min D}
               end) Ds}
```

choose

```
{FD.choose +Dist +Xv ?X ?Spec}
```

Chooses the element `X` in `Xv` according to the description `Dist`. A specification `Spec` for the element `X` is returned according to the description `Dist`. The parameter `Dist` is defined in the same way as for `FD.distribute` except for the value selection. If the feature `value` is used for generic distribution, the field must be constrained to a unary function `P` which selects a value from the domain of the selected variable (see below for an example). For example,

```
{FD.choose ff Xs E S}
```

selects the element `E` in the list `Xs` according to the first-fail strategy and binds `S` to the current lower bound of `E`.

```
{FD.choose generic(value:splitMin) Xv E S}
```

selects the element `E` in the list `Xs` according to the first-fail strategy and binds `S` to the pair `0#M`, where `M` is the result of `{FD.reflect.mid E}`. For the naive distribution strategy, the following may be used.

```
{FD.choose generic(value: fun{$ X}
                             {FD.reflect.min X}
                          end)
 Xv E S}
```

5.13 Assigning Values

Special support is available for assigning particular values to vectors of variables. Assignment interleaves the assignment of a value proper to a variable and synchronization until stability after each assignment.

The selection of variables and the selection of values is as with distribution Section 5.12.

assign

```
{FD.assign +ValA +Xv}
```

The vector `Xv` is assigned according to the specification `ValA`. `ValA` may be either the atom `min`, `mid`, or `max`. That is, the smallest, medium, or largest element is assigned to each variable.

Is equivalent to (for a list of variables `Xs`):

```
proc {FD.assign ValA Xs}
   for X in Xs do
      {Space.waitStable}
      X = {FD.reflect.ValA X}
   end
end
```

Scheduling Support: `Schedule`

This chapters describes propagators and distributors for scheduling applications. More information on scheduling in Oz can be found in [10] and [11]. A tutorial account on scheduling can be found in Chapter *Scheduling, (Finite Domain Constraint Programming in Oz. A Tutorial.)*.

6.1 Serialization for Unary Resources

Serializing a unary resource which can execute at most one task simultaneously means that the tasks must be scheduled non-overlapping in time.

The following conventions hold. The argument *StartR* is a record of finite domain integers denoting start times of tasks. The argument *DurR* is a record of integers denoting durations of tasks. The arities of *StartR* and *DurR* must be equal.

The integers and literals occurring in *TasksLIvv* denote the tasks to be scheduled. Each element of *TasksLIvv* must occur in the arity of *StartR*. The tasks occurring in the vectors *TasksLIv* are scheduled on the same resource.

`serializedDisj`

```
{Schedule.serializedDisj +TasksLIvv +StartR +DurR}
```

creates a propagator, which states that all tasks *TasksLIv* scheduled on the same resource must not overlap in time.

The propagator does the same propagation as the conjunction of all reified constraints modelling that two tasks must not overlap in time, i.e.

```
(StartR.T1 + DurR.T1 =<: StartR.T2) +
(StartR.T2 + DurR.T2 =<: StartR.T1) =: 1
```

where *T1* and *T2* are two tasks out of *TasksLIvv*.

Assume the following tasks and durations:

Task	Resource	Duration
a	r	4
b	r	6
c	r	7
d	s	7
e	s	4

In addition let us assume that no further restriction on the start times is given.

Then

```
Dur   = dur(a:4 b:6 c:7 d:7 e:4)
Start = {FD.record start [a b c d e] 0#FD.sup}
Tasks = (a#b#c)#(d#e)
{Schedule.serializedDisj Tasks Start Dur}
```

serializes the tasks for the resources `r` and `s` (for `FD.record` see (page 35)). Note that the resources are kept anonymous, they are just reflected by the vector elements in `Tasks`. If we would like to make the resources more explicit we could use for `Tasks` the following:

```
Tasks = tasks(r:[a b c] s:[d e])
```

It also possible to use integers or names rather than atoms for the tasks.

`serialized`

`{Schedule.serialized +TasksLIvv +StartR +DurR}`

creates a propagator, which states that all tasks *TasksLIv* scheduled on the same resource must not overlap in time.

The propagator does stronger propagation than `Schedule.serializedDisj` by using so-called edge-finding. This type of edge-finding is a generalization of a technique described in [4].

`taskIntervals`

`{Schedule.taskIntervals +TasksLIvv +StartR +DurR}`

creates a propagator, which states that all tasks *TasksLIv* scheduled on the same resource must not overlap in time.

The propagator does even stronger propagation than `Schedule.serialized` by using so-called task-intervals [2]. The propagation of this propagator is slightly weaker than the propagation described in [2].

6.2 Distribution

In addition to the conventions used in Section 6.1 the record of start times *StartR* must have the feature `pe`. This feature denotes the task which is to be scheduled last, i.e. the makespan of the schedule.

`firstsDist`

`{Schedule.firstsDist +TasksLIvv +StartR +DurR}`

distributes the tasks occurring in *TasksLIvv*, such that every resource is serialized.

More details can be found in [1].

`lastsDist`

`{Schedule.lastsDist +TasksLIvv +StartR +DurR}`

distributes the tasks occurring in *TasksLIvv*, such that every resource is serialized.

More details can be found in [1].

`firstsLastsDist`

```
{Schedule.firstsLastsDist +TasksLIvv +StartR +DurR}
```

distributes the tasks occurring in *TasksLIvv*, such that every resource is serialized.

More details can be found in [1].

`taskIntervalsDistP`

```
{Schedule.taskIntervalsDistP +TasksLIvv +StartR
                             +DurR}
```

distributes the tasks occurring in *TasksLIvv*, such that every resource is serialized. This strategy is well suited for proving optimality.

More details can be found in [2]. The distribution strategy implemented in Oz differs slightly from the one described in [2].

`taskIntervalsDistO`

```
{Schedule.taskIntervalsDistO +TasksLIvv +StartR
                             +DurR}
```

distributes the tasks occurring in *TasksLIvv*, such that every resource is serialized. This strategy is well suited for finding good solutions in combination with local search techniques.

More details can be found in [2]. The distribution strategy implemented in Oz differs slightly from the one described in [2].

6.3 Cumulative Scheduling

The following conventions hold. The argument *StartR* is a record of finite domain integers denoting start times of tasks. The argument *DurR* is a record of integers denoting durations of tasks. The argument *UseR* is a record of integers denoting the resource usage of tasks. The arities of *StartR*, *DurR*, and *UseR* must be equal.

The integers and literals occurring in *TasksLIvv* denote the tasks to be scheduled. Each element of *TasksLIvv* must occur in the arity of *StartR*. The tasks occurring in the vectors *TasksLIv* are scheduled on the same resource. The vector *CapIv* is a vector of integers denoting the capacity of the resources. The number of elements in the vectors *TasksLIvv* and *CapIv* must be equal.

`cumulative`

```
{Schedule.cumulative +TasksLIvv +StartR +DurR
                     +UseR +CapIv}
```

creates a propagator, which states that for all resources i and time instants x, the resource usage does not exceed the available capacity:

$$\sum_{\{t \in \mathrm{TasksA}_i \mid \mathrm{StartR}.t \leq x < \mathrm{StartR}.t + \mathrm{DurR}.t\}} \mathrm{UseR}.t \leq \mathrm{CapI}_i$$

The propagator does not use edge-finding.

Assume that we have the following resources and tasks:

Resource	Capacity
r	5
s	2

Task	Resource	Duration	Usage
a	r	5	**5**
b	r	2	**3**
c	s	7	**2**
d	s	4	**3**
e	s	9	**5**

Provided that no limit on the start times of the tasks are given, the following

```
Tasks = tasks([a b] [c d e])
Start = {FD.record start [a b c d e] 0#FD.sup}
Dur   = dur(a:5 b:2 c:7 d:4 e:9)
Use   = use(a:5 b:3 c:2 d:3 e:5)
Cap   = cap(5 2)
{Schedule.cumulative Tasks Start Dur Use Cap}
```

propagates that the resource usage does not exceed the resources' capacities (for `FD.record` see (page 35)).

`cumulativeEF`

`{Schedule.cumulativeEF +TasksLIvv +StartR +DurR +UseR +CapIv}`

creates a propagator, which states that for all resources i and time instants x, the resource usage does not exceed the available capacity:

$$\sum_{\{t \in \text{TasksA}_i | \text{StartR}.t \leq x < \text{StartR}.t + \text{DurR}.t\}} \text{UseR}.t \leq \text{CapI}_i$$

This propagator generalizes the edge-finding propagation in `Schedule.serialized` to deal with non-unary resources.

`cumulativeTI`

`{Schedule.cumulativeTI +TasksLIvv +StartR +DurR +UseR +CapIv}`

creates a propagator, which states that for all resources i and time instants x, the resource usage does not exceed the available capacity:

$$\sum_{\{t \in \text{TasksA}_i | \text{StartR}.t \leq x < \text{StartR}.t + \text{DurR}.t\}} \text{UseR}.t \leq \text{CapI}_i$$

This propagator generalizes the edge-finding propagation in `Schedule.taskIntervals` to deal with non-unary resources.

`cumulativeUp`

`{Schedule.cumulativeUp +TasksLIvv +StartR +DurR +UseR +CapIv}` creates a propagator, which states that for all resources i and time instants x, the resource is at least as large as the available capacity:

$$\sum_{\{t \in \mathrm{TasksA}_i | \mathrm{StartR}.t \leq x < \mathrm{StartR}.t + \mathrm{DurR}.t\}} \mathrm{UseR}.t \geq \mathrm{CapI}_i$$

6.4 Miscellaneous Propagators

disjoint

```
{Schedule.disjoint *D1 +I1 *D2 +I2}
```

creates a propagator for $D1 + I1 \leq D2 \ \vee \ D2 + I2 \leq D1$. Its operational semantics is defined by

```
or D1 + I1 =<: D2
[] D2 + I2 =<: D1
end
```

7

Finite Set Constraints: FS

We use the following notation for operations and relations on sets. We write $\cup, \cap$, and $\setminus$ for set union, intersection, and difference, $\subseteq$ and $\parallel$ for inclusion and disjointness, # for the set cardinality, and $\in$ for the element relation. Furthermore, we write $\emptyset$ and $\mathcal{U}$ for the empty set and the universal set.

For every set specification *Spec* we write the set *M* specified by *Spec* as *M* =

set(*Spec*). For example, *set*(`[1#3 5 7]`) denotes $\{1,2,3,5,7\}$. Further, for every set `S` we denote with $D = set^{-1}(S)$ a set description *D* such that *set*(`D`) $=$ `S`.

For more information on the finite set constraint system see [5].

7.1 Finite Set Intervals

`inf`

```
FS.inf
```

An integer constant that denotes the smallest possible element of a set. Its value is implementation-dependent. In Mozart `FS.inf` is 0.

`sup`

```
FS.sup
```

An integer constant that denotes the greatest possible element of a set. Its value is implementation-dependent. In Mozart `FS.sup` is 134 217 726.

`compl`

```
{FS.compl $M1 $M2}
```

$\texttt{M2} = \{\texttt{FS.inf},\ldots,\texttt{FS.sup}\} \setminus \texttt{M1}$

`complIn`

```
{FS.complIn $M1 $M2 $M3}
```

$\texttt{M3} = \texttt{M2} \setminus \texttt{M1}$

`include`

```
{FS.include +D *M}
```

$\texttt{D} \in \texttt{M} \wedge \texttt{FS.inf} \leq \texttt{D} \leq \texttt{FS.sup}$

`exclude`

```
{FS.exclude +D *M}
```

$D \notin M$

card

```
{FS.card *M ?D}
```

$D = \#M$

cardRange

```
{FS.cardRange +I1 +I2 *M}
```

$I1 \leq \#M \leq I2$

isIn

```
{FS.isIn +I *M ?B}
```

$(E \in M) \to B$

makeWeights

```
{FS.makeWeights +SpecW ?P}
```

Returns a procedure with signature `{P +I1 ?I2}`. This procedure maps an element to a weight according to the weight description passed to `FS.makeWeights`.

7.2 Sets over Integers

int.min

```
{FS.int.min *M $D}
```

`D` is the minimal element within `M`.

int.max

```
{FS.int.max *M $D}
```

`D` is the maximal element within `M`.

int.convex

```
{FS.int.convex *M}
```

Whenever `I1` and `I2` are elements of `M`, then every `I` between `I1` and `I2`, `I1 < I < I2`, is also in `M`.

int.match

```
{FS.int.match *M *Dv}
```

`Dv` is a vector of integer variables that denotes the elements of `M` in ascending order.

int.minN

```
{FS.int.minN *M *Dv}
```

`Dv` is a vector of *n* integer variables that denotes the *n* minimal elements of `M` in ascending order.

int.maxN

```
{FS.int.maxN *M *Dv}
```

`Dv` is a vector of *n* integer variables that denotes the *n* maximal elements of `M` in ascending order.

int.seq

{FS.int.seq *Mv}

Mv is a vector of disjoint sets such that for distinct sets M1 and M2, where M1 precedes M2 in Mv, all elements of M1 are smaller than any element of M2.

7.3 Standard Propagators

diff

{FS.diff $M1 $M2 $M3}

$\mathtt{M3} = \mathtt{M1} \setminus \mathtt{M2}$

intersect

{FS.intersect $M1 $M2 $M3}

$\mathtt{M3} = \mathtt{M1} \cap \mathtt{M2}$

intersectN

{FS.intersectN *Mv *M}

$\mathtt{M} = \bigcap \{\mathtt{M'} \mid \mathtt{M'} \in \mathtt{Mv}\}$

union

{FS.union $M1 $M2 $M3}

$\mathtt{M3} = \mathtt{M1} \cup \mathtt{M2}$

unionN

{FS.unionN $Mv $M}

$\mathtt{M} = \bigcup \{\mathtt{S} \mid \mathtt{S} \in \mathtt{Mv}\}$

subset

{FS.subset $M1 $M2}

$\mathtt{M1} \subseteq \mathtt{M2}$

disjoint

{FS.disjoint $M1 $M2}

$\mathtt{M1} \| \mathtt{M2}$

disjointN

{FS.disjointN *Mv}

All elements of the vector *Mv* are pairwise disjoint.

distinct

{FS.distinct $M1 $M2}

$\mathtt{M1} \neq \mathtt{M2}$

distinctN

{FS.distinctN *MV}

All elements of the vector *Mv* are pairwise distinct.

partition

```
{FS.partition $MV $M}
```

Mv is a partition of *M*; that is, *Mv* contains pairwise disjoint sets such that their union yields *M*.

7.4 Finite Set Interval Variables

var.is

```
{FS.var.is +M ?B}
```

Tests whether `M` is a finite set variable.

7.4.1 Declaring a Single Variable

var.decl

```
{FS.var.decl ?M}
```

$\emptyset \subseteq \mathtt{M} \subseteq \mathcal{U}$

var.upperBound

```
{FS.var.upperBound +Spec ?M}
```

$\emptyset \subseteq \mathtt{M} \subseteq set(\mathtt{Spec})$

var.lowerBound

```
{FS.var.lowerBound +Spec ?M}
```

$set(\mathtt{Spec}) \subseteq \mathtt{M} \subseteq \mathcal{U}$

var.bounds

```
{FS.var.bounds +Spec1 +Spec2 ?M}
```

$set(\mathtt{Spec1}) \subseteq \mathtt{M} \subseteq set(\mathtt{Spec2})$

7.4.2 Declaring a List of Variables

The following functions return a list `Ms` of length `I` and all its elements are constrained to finite set interval variables according to the following specifications.

var.list.decl

```
{FS.var.list.decl +I ?Ms}
```

For all elements `M` of `Ms`: $\emptyset \subseteq \mathtt{M} \subseteq \mathcal{U}$

var.list.upperBound

```
{FS.var.list.upperBound +I +Spec ?Ms}
```

For all elements `M` of `Ms`: $\emptyset \subseteq \mathtt{M} \subseteq set(\mathtt{Spec})$

var.list.lowerBound

```
{FS.var.list.lowerBound +I +Spec ?Ms}
```

For all elements `M` of `Ms`: $set(\mathtt{Spec}) \subseteq \mathtt{M} \subseteq \mathcal{U}$

var.list.bounds

```
{FS.var.list.bounds +I +Spec1 +Spec2 ?Ms}
```

For all elements `M` of `Ms`: $set(\mathtt{Spec1}) \subseteq \mathtt{M} \subseteq set(\mathtt{Spec2})$

7.4.3 Declaring a Tuple of Variables

The following functions return a tuple `Mt` with label `L` and width `I` and all its elements are constrained to finite set interval variables according to the following specifications.

`var.tuple.decl`

```
{FS.var.tuple.decl +L +I ?Mt}
```

For all elements `M` of `Mt`: $\emptyset \subseteq \mathtt{M} \subseteq \mathcal{U}$

`var.tuple.upperBound`

```
{FS.var.tuple.upperBound +L +I +Spec ?Mt}
```

For all elements `M` of `Mt`: $\emptyset \subseteq \mathtt{M} \subseteq \mathit{set}(\mathtt{Spec})$

`var.tuple.lowerBound`

```
{FS.var.tuple.lowerBound +L +I +Spec ?Mt}
```

For all elements `M` of `Mt`: $\mathit{set}(\mathtt{Spec}) \subseteq \mathtt{M} \subseteq \mathcal{U}$

`var.tuple.bounds`

```
{FS.var.tuple.bounds +L +M +Spec1 +Spec2 ?Mt}
```

For all elements `M` of `Mt`: $\mathit{set}(\mathtt{Spec1}) \subseteq \mathtt{M} \subseteq \mathit{set}(\mathtt{Spec2})$

7.4.4 Declaring a Record of Variables

The following functions return a record `Mr` with label `L` and the fields `Ls` and all its fields are constrained to finite set interval variables according to the following specifications.

`var.record.decl`

```
{FS.var.record.decl +L +Ls ?Mr}
```

For all elements `M` of `Mr`: $\emptyset \subseteq \mathtt{M} \subseteq \mathcal{U}$

`var.record.upperBound`

```
{FS.var.record.upperBound +L +Ls +Spec ?Mr}
```

For all elements `M` of `Mr`: $\emptyset \subseteq \mathtt{M} \subseteq \mathit{set}(\mathtt{Spec})$

`var.record.lowerBound`

```
{FS.var.record.lowerBound +L +Ls +Spec ?Mr}
```

For all elements `M` of `Mr`: $\mathit{set}(\mathtt{Spec}) \subseteq \mathtt{M} \subseteq \mathcal{U}$

`var.record.bounds`

```
{FS.var.record.bounds +L +Ls +Spec1 +Spec2 ?Mr}
```

For all elements `M` of `Mr`: $\mathtt{M} \in [\mathit{set}(\mathtt{Spec1}), \mathit{set}(\mathtt{Spec2})]$

7.5 Finite Set Constants

`value.empty`

```
FS.value.empty
```

Denotes $\emptyset$.

value.universal

```
FS.value.universal
```

Denotes u.

value.singl

```
{FS.value.singl +I ?M}
```

$\mathtt{M} = \{\mathtt{I}\}$

value.make

```
{FS.value.make +Spec ?M}
```

$\mathtt{M} = \mathit{set}(\mathtt{Spec})$

value.is

```
{FS.value.is +M ?B}
```

Tests whether `M` is a finite set value or not.

value.toString

```
{FS.value.toString +M ?S}
```

Converts `M` to a string and returns it in `M`.

7.6 Reified Propagators

reified.isIn

```
{FS.reified.isIn +I *M $D}
```

$\mathtt{D} \in \{0,1\} \wedge ((\mathtt{I} \in \mathtt{M}) \leftrightarrow \mathtt{D} = 1)$

reified.areIn

```
{FS.reified.areIn +Spec *M $Ds}
```

Spec describes a list of individual elements *Is*. *Is* and *Ds* are lists of the same length such that every element $\mathtt{D}_i$ of *Ds* reifies the presence of the corresponding element $\mathtt{I}_i$ of *Is* in the set *M*.

reified.include

```
{FS.reified.include +D1 *M $D2}
```

D2 reifies the presence of *D1* in the set *M*. This propagator detects in contrast to `FS.reified.isIn` earlier if *D1* is or is not constained in *M*.

reified.equal

```
{FS.reified.equal *M1 *M2 $D}
```

D reifies the equality of *M1* and *M2*.

reified.partition

```
{FS.reified.partition +MVs +Is +MV $Ds}
```

The propagator partitions the set value `MV` by selecting elements from the list of set values `MVs`. The positive integers in `Is` denote the cost resp. benefit of the corresponding set value in `MVs` if it is selected for the partition. Each element of `Ds` is either `0` or the corresponding integer value in `Is` depending on whether the corresponding set value in `MVs` is part of the partition or not. Excluding a set value from the partition is done by constraining the corresponding element of `Ds` to `0`. An element in `Ds` not equal to `0` includes the corresponding set value in `MVs` in the partition. The propagator ensures a valid partition according to the values of `Ds`.

7.7 Iterating and Monitoring

`monitorIn`

```
{FS.monitorIn *M ?Is}
```

This procedure writes all elements of `M` to `Is` as soon as $\texttt{I} \in \texttt{M}$ becomes *known*. When `M` becomes determined the stream `Is` will be closed.

`monitorOut`

```
{FS.monitorOut *M ?Is}
```

This procedure writes all elements of `M` to `Is` as soon as $\texttt{I} \notin \texttt{M}$ becomes *known*. When `M` becomes determined the stream `Is` will be closed.

`forAllIn`

```
{FS.forAllIn *M +P/1}
```

This procedure applies `P/1` to all elements of `M`.

7.8 Reflection

The result of a reflective procedure depends on the current state of the constraint store and is non-deterministic.

`reflect.card`

```
{FS.reflect.card *M ?Spec}
```

returns a description *Spec* of the current information on the cardinality of M.

`reflect.lowerBound`

```
{FS.reflect.lowerBound *M ?Spec}
```

Returns a specification of the greatest lower bound that is currently known about the set M.

`reflect.upperBound`

```
{FS.reflect.upperBound *M ?Spec}
```

Returns a specification of the least upper bound that is currently known about the set M.

`reflect.unknown`

```
{FS.reflect.unknown *M ?Spec}
```

Returns a specification of the set of elements that are neither known to be included in M nor excluded from M.

`reflect.lowerBoundList`

```
{FS.reflect.lowerBoundList *M ?Spec}
```

Returns an expanded specification (i.e., every individual element is represented) of the greatest lower bound that is currently known about the set *M*.

reflect.upperBoundList

```
{FS.reflect.upperBoundList *M ?Spec}
```

Returns an expanded specification (i.e., every individual element is represented) of the least upper bound that is currently known about the set *M*.

reflect.unknownList

```
{FS.reflect.unknownList *M ?Spec}
```

Returns an expanded specification (i.e., every individual element is represented) of the set of elements that are neither known to be included in *M* nor excluded from *M*.

reflect.cardOf.lowerBound

```
{FS.reflect.cardOf.lowerBound *M ?I}
```

Returns the cardinality of the current greatest lower bound for *M*.

reflect.cardOf.upperBound

```
{FS.reflect.cardOf.upperBound *M ?I}
```

Returns the cardinality of the current greatest lower bound for *M*.

reflect.cardOf.unknown

```
{FS.reflect.cardOf.unknown *M ?I}
```

Returns the number of elements that are currently not known to be included or excluded from *M*.

7.9 Distribution

Given a set *M*, let $lowerBound(\texttt{M})$ and $upperBound(\texttt{M})$ denote the greatest lower bound and the least upper bound currently known for *M*. Also define $unknown(\texttt{M}) = upperBound(\texttt{M}) \setminus lowerBou$

distribute

```
{FS.distribute +Dist *Ms}
```

The vector `Ms` is distributed according to the specification `Dist`. The following values for `Dist` are supported:

- `naive` is equivalent to `generic`, i.e. the default settings apply.
-
```
generic(order:    +Order   <= order
        filter:   +Filter  <= true
        select:   +Select  <= id
        element:  +Element <= element
        rrobin:   +RRobin  <= false
        weights:  +Weights <= {FS.makeWeights nil}
        procedure:+Proc    <= proc {$} skip end)
```

  - `Order`
    - `naive` selects the left-most variable.

    *
      ```
      order(sel:  +Sel  <= min
            cost: +Cost <= card
            comp: +Comp <= unknown)
      ```
      · `Sel = min` selects the left-most variable `S` from `Ss` with the *minimal* cost according to `Cost`.
      · `Sel = max` selects the left-most variable `S` from `Ss` with the *maximal* cost according to `Cost`.
      · `Cost = card`: The cost is the cardinality of the set determined by `Comp`.
      · `Cost = weightSum`: The cost is the *sum* of the weights associated with the elements of the set determined by `Comp`.
      · `Cost = weightMin`: The cost is the *minimal* weight determined by `Comp`.
      · `Cost = weightMax`: The cost is the *maximal* weight associated with an element of the set determined by `Comp`.
      · `Comp = unknown` selects *unknown*(`S`).
      · `Comp = lowerBound` selects *lowerBound*(`S`).
      · `Comp = upperBound` selects *upperBound*(`S`).
    * `fun {Order +Ss} ... end`
  - `Filter` determines if an element `S` of `Ss` is choosen for distribution. That is the case if `{IsDet S}` *and* the filter yields `true`.
    * `true` skips values in `Ss`.
    * `fun {Filter +E} ... end`
  - `Select` is used to access the actual finite set variable. Self-defined functions resp. procedures have to apply an appropriate selection function by themselves.
    * `id` is the identity function.
    * `fun {Select +E} ... end`
  - `Element`

    *
      ```
      element(sel: +Sel <= min
              wrt: +Wrt <= unknown)
      ```
      · `Sel = min` selects the *minimal* element with respect to `Wrt`.
      · `Sel = max` selects the *maximal* element with respect to `Wrt`.
      · `Wrt = unknown` chooses an element from *unknown*(S). and interprets it as an integer.
      · `Wrt = weight` chooses an element from *unknown*(S) and takes its weight as selection criterion.
    * `fun {Element +E} ... end`
  - `RRobin`
    * `true` causes the distribution to step through the variable list in a round-robin fashion.
    * `false` causes the distribution to completely enumerate the head of the variable list and then proceeds with the head of the tail of the variable list.

- `Weights` must be a list of the form `[E#W ...]`. The variable `E` denotes an element and `W` the element's weight. An list element of the form `default#W` assigns to all not explicitely mentioned elements the weight `W`. If there is no element `default#W` then `default#0` is implicitely added.
- `Proc` is applied when stability is reached. Since this application may cause instability, distribution is continued when stability is reached again.

8

Feature Constraints: RecordC

This chapter explains procedures dedicated to feature constraints.

is

```
{RecordC.is *X ?B}
```

tests whether X has kind record.

tell

```
{RecordC.tell +L ?R}
```

tells the constraint store that R is a record with label L.

tellSize

```
{RecordC.tellSize +L +I ?R}
```

tells the constraint store that R is a record with label L.

Signals the implementation that it is likely that I features are told to R. RecordC.tellSize is semantically equivalent to RecordC.tell, but the current implementation optimizes memory allocation.

^

```
{RecordC.'^' R +LI X}
```

tells the constraint store that R is a FC having field X at feature LI.

Is supported by the infix operator ^, that is

```
{RecordC.'^' R LI X}
```

can also be written as

```
R^LI=X
```

width

```
{RecordC.width *R ?D}
```

posts a propagator for the constraint that D is the width of R. Also tells the constraint store that D is a finite domain integer.

reflectArity

```
{RecordC.reflectArity *R ?LIs}
```

returns a list LIs containing the currently known features of R.

`monitorArity`

`{RecordC.monitorArity *R ?P ?LIs}`

returns a nullary procedure `P` and a stream `LIs` containing the currently known features of `R`.

Features appear in the stream as soon as they become known to the constraint store. Application of `P` closes the stream and deletes the propagator. The stream is automatically closed once the constraint store determines `R`.

`hasLabel`

`{RecordC.hasLabel *R ?B}`

blocks until `R` becomes a feature structure. Tests whether `R` has been told a label with `RecordC.tell`.

9

Deep-guard Concurrent Constraint Combinators: `Combinator`

This chapter describes deep-guard concurrent constraint combinators such as conditional and disjunction. Most combinators implemented by the module `Combinator` are available by convenient syntax and are described in Chapter *Logic Programming, (Tutorial of Oz)*.

`'not'`

```
{Combinator.'not' +P}
```

implements deep-negation where the nullary procedure `P` gives the statement to negate.

Is supported by special syntax. The statement

```
not S end
```

expands to

```
{Combinator.'not' proc {$} S end}
```

`'reify'`

```
{Combinator.'reify' +P $D}
```

implements deep-reification where the nullary procedure `P` gives the statement to reify.

`'cond'`

```
{Combinator.'cond' +T +P}
```

implements parallel concurrent conditional.

`'or'`

```
{Combinator.'or' +T}
```

implements disjunction.

`'choice'`

```
{Combinator.'choice' +T}
```

implements choice point construction.

`'dis'`

```
{Combinator.'dis' +T}
```

implements andorra-style disjunction.

First-class Computation Spaces: Space

First-class computation spaces can be used to program inference engines for problem solving.

is

    {Space.is +X ?B}

tests whether X is a space.

new

    {Space.new +P ?Space}

returns a newly created space, in which a thread containing an application of the unary procedure P to the root variable of Space is created.

ask

    {Space.ask +Space ?T}

waits until Space becomes stable or merged and then returns the status of Space.

If Space is merged, the atom merged is returned.

If Space is stable and:

failed	the atom failed is returned.
succeeded	and there are no threads in Space synchronizing on choices, the atom succeeded is returned.
succeeded	and there is at least one thread in Space which synchronizes on a choice the tuple alternatives(*I*) is returned, where *I* gives the number of alternatives of the selected choice.

Synchronizes on stability of Space.

Raises a runtime error if the current space is not admissible for Space.

askVerbose

    {Space.askVerbose +Space ?T}

returns the status of `Space` in verbose form. Reduces immediately, even if `Space` is not yet stable.

If `Space` becomes merged, the atom `merged` is returned.

If `Space` becomes suspended (that is, blocked but not stable), `T` is bound to the tuple `suspended(`*T1*`)`. *T1* is a future that is bound to the status of `Space` when `Space` becomes unblocked again.

If `Space` is stable and:

failed the atom `failed` is returned.

succeeded and there are no threads in `Space` synchronizing on choices, the tuple `succeeded(`*A*`)` is returned. The atom *A* is either `stuck`, when `Space` still contains threads, or `entailed` otherwise.

succeeded and there is at least one thread in `Space` which synchronizes on a choice the tuple `alternatives(`*I*`)` is returned, where `I` gives the number of alternatives of the selected choice.

Does not synchronize on stability of `Space`.

Raises a runtime error if the current space is not admissible for `Space`.

`merge`

```
{Space.merge +Space X}
```

merges `Space` with the current space and constrains `X` to the root variable of `Space`.

Does not synchronize on stability of `Space`.

Raises a runtime error if `Space` is already merged, or if the current space is not admissible for `Space`.

`clone`

```
{Space.clone +Space1 ?Space2}
```

blocks until `Space1` becomes stable and returns a new space which is a copy of `Space1`.

Synchronizes on stability of `Space`.

Raises a runtime error if `Space` is already merged, or if the current space is not admissible for `Space`.

`inject`

```
{Space.inject +Space +P}
```

creates a thread in the space `Space` which contains an application of the unary procedure `P` to the root variable of `Space`.

Does not synchronize on stability of `Space`.

Raises a runtime error if `Space` is already merged, or if the current space is not admissible for `Space`.

`kill`

```
{Space.kill +Space}
```

kills a space by injecting failure into `Space`.

Can be defined by

```
proc {Space.kill S}
   {Space.inject S proc {$ _} fail end}
end
```

Does not synchronize on stability of `Space`.

Raises a runtime error if `Space` is already merged, or if the current space is not admissible for `Space`.

`commit`

`{Space.commit +Space +IT}`

blocks until `Space` becomes stable and then commits to alternatives of the selected choice of `Space`.

If `IT` is a pair of integers *l*#*r* then all but the *l*, *l* + 1, . . . , *r* alternatives of the selected choice of `Space` are discarded. If a single alternative remains, the topmost choice is replaced by this alternative. If no alternative remains, the space is failed.

An integer value for `IT` is an abbrevation for the pair `IT#IT`.

Synchronizes on stability of `Space`.

Raises a runtime error, if `Space` has been merged already, if there exists no selected choice in `Space`, or if the current space is not admissible for `Space`.

`waitStable`

`{Space.waitStable}`

blocks until the current space (the space that hosts the current thread) becomes stable. `Space.waitStable` is used mainly for programming distribution strategies (see for example Chapter *User-Defined Distributors, (Finite Domain Constraint Programming in Oz. A Tutorial.)*), where for accurate variable selection it is required that all propagation has been carried out.

If executed in the toplevel space, it will block forever.

`choose`

`{Space.choose +I1 ?I2}`

blocks until the current space becomes stable. When the current space becomes stable it creates a choice point with `I1` alternatives. `I2` is bound to the value selected by `Space.commit`. Reduces as soon as `I2` becomes bound.

`Space.choose` is a primitive intended for programming abstractions. For example, the Mozart-compiler expands the following `choice`-statement

```
choice S1 [] S2 end
```

to the following statement

```
case {Space.choose 2}
of 1 then S1
[] 2 then S2
end
```

If executed in the toplevel space, it will block forever.

Part III

Distributed Programming

Connecting Computations: `Connection`

Oz uses a ticket-based mechanism to establish connections between independent Oz processes. One process (called server) creates a ticket with which other sites (called clients) can establish a connection.

A ticket is a character string which can be stored and transported through all media that can handle text, e.g., phone lines, electronic mail, paper, and so forth. Tickets are insecure, in that they can be forged (albeit some luck is required, since they offer some security against typos). The following is an example ticket encoded as an Oz atom:

```
'x-ozticket://134.96.186.115:9000:egbj0:DS/v:s:kn'
```

The ticket identifies both the server and the value to which a remote reference will be made. Independent connections can be made to different values on the same server. Once an initial connection is established by the value exchanged, then further connections as desired by applications can be built from programming abstractions, like object, classes, ports or procedures.

Two different types of connections and tickets are supported:

One-to-one connections and one-shot tickets. Allow to establish a single connection only.

Many-to-one connections and many-shot tickets. Allow multiple connections with the same ticket to the same value. Values for many-to-one connections are offered through gates.

The module `Connection` provides procedures and classes that support both one-to-one and many-to-one connections.

11.1 One-to-one Connections

`offer`

```
{Connection.offer X TicketA}
```

Returns the single-shot ticket `TicketA` (an atom) under which the value `X` is offered.

The value `X` is exported immediately. An exception is raised, if exportation of `X` fails, because `X` refers to sited entities.

take

`{Connection.take +TicketV X}`

Returns the value X offered under the ticket TicketV (a virtual string).

Waits until the connection to the offering process is established and the ticket has been acknowledged by that process.

Raises an exception if the ticket is illegal, or if the offering process does not longer exist.

Also works for many-shot tickets, where an exception might be raised if the same ticket is used more than once.

11.2 Many-to-one Connections

Values for many-to-one connections can be offered through gates. Values offered through gates can be taken with `Connection.take` as described for one-to-one connections. Gates can be closed which implies that the associated ticket becomes invalid.

The special case and frequently occuring case that the gate never gets closed is supported by the procedure `Connection.offerUnlimited` which is as follows:

offerUnlimited

`{Connection.offerUnlimited X ?TicketA}`

offers the value X under the returned many-shot ticket TicketA (an atom).

The value X is exported immediately. An exception is raised, if exportation of X fails, because X might refer to sited entities.

The ticket remains valid until the current Oz process terminates.

Gates are provided as instances of the class `Connection.gate`. The methods of `Connection.gate` are as follows.

init

`init(X TicketA <= _)`

Optionally returns the many-shot ticket TicketA (an atom) under which the value X is offered.

The value X is exported immediately. An exception is raised, if exportation of X fails, because X might refer to sited entities.

getTicket

`getTicket(TicketA)`

Returns the many-shot ticket TicketA (an atom) of the gate.

close

`close()`

Closes the gate, which makes further use of the associated ticket illegal.

Spawning Computations Remotely: Remote

The module `Remote` provides the class `Remote.manager` by which new Oz processes on arbitrary networked computers that also have Mozart installed can be created. Creating an instance of that class does the following two things:

- A new Oz process with a module manager `M` is created on a networked computer.
- The newly created object `O` serves as a proxy to `M`. `O` is called a remote module manager. This allows to start applications remotely that access remote resources by local system modules.

The methods of the class `Remote.manager` are as follows.

init

```
init(host:   +HostV   <= localhost
     fork:   +ForkA   <= automatic
     detach: +DetachB <= false)
```

Creates a new Oz process at the computer with host name `HostV` (specified by a virtual string), where `localhost` is the computer running the current Oz process.

`ForkA` (an atom) determines an operating system command to fork the remote Oz process. The atoms `'automatic'` and `'sh'` have special meaning. `'automatic'` is the default method. Other useful values for `ForkA` are `'rsh'` (remote shell command) and `'ssh'` (secure shell command).

`sh` configuration If `'sh'` is used as fork method, a new Oz engine is created on the current host by using the Unix `sh` command. You can test whether this method works on your computer by:

```
sh -c 'ozengine x-oz://system/RemoteServer.ozf --test'
```

This should be always the case, if Mozart has been installed properly on your computer. This in particular requires that `$OZHOME/bin` is in your path of executables (`$OZHOME` refers to the directory where Mozart has been installed).

Note that the value of `HostV` is ignored (the hostname is assumed to be `localhost`), if `'sh'` is used as fork method.

If `HostV` is `'localhost'` and `ForkA` is `'automatic'` (which is the default), then on some platforms the forked and forking processes communicate through shared memory rather than sockets, which is more efficient. The system property `'distribution.virtualsites'` carries a boolean telling whether this facility, called virtual sites, is supported in the running Mozart process; the `'sh'` fork method is used as a fall-back.

If `HostV` is different from `'localhost'` and the method is `'automatic'` the command `'rsh'` is used. `'rsh'` creates a shell remotely by using the Unix `rsh` command, which in turn creates the new Oz engine.

`rsh` configuration Remote managers with method `rsh` only work properly, if the `rsh` command has been set up properly. You can test it for the host *Host* by executing the following command in the operating system shell:

```
rsh Host ozengine x-oz://system/RemoteServer.ozf --test
```

If the message

```
Remote: Test succeeded...
```

is printed, your configuration is okay. This requires two things:

1. Execution of `rsh` *Host* must not prompt for a password. This is usually achieved by having a special file `.rhosts` in your home directory. Each entry in that file must be a host name. For those hosts having an entry in that file, `rsh` does not prompt for a password.

   Take the following sample `.rhosts` file at the computer `wallaby.ps.uni-sb.de`:

   ```
   godzilla.ps.uni-sb.de
   bamse.sics.se
   ```

   If `rsh wallaby.ps.uni-sb.de` is executed on `bamse.sics.se` or `godzilla.ps.uni-sb.de`, then `rsh` does not prompt for a password.

   With other words, all host names that you ever want to use with `Remote.manager` should be in `.rhosts`.

2. After the login performed by `rsh` the command `ozengine` must be executable. This should be always the case, if Mozart has been installed properly on your computer. This in particular requires that `$OZHOME/bin` is in your path of executables (`$OZHOME` refers to the directory where Mozart has been installed).

Other commands Rather than using `rsh`, any value for `ForkA` is possible. In that case the following operating system command:

```
ForkA Host ozengine x-oz://system/RemoteServer.ozf --test
```

should print the message

```
Remote: Test succeeded...
```

A prominent example of a different command and a very recommended substitute for `rsh` is `ssh` (secure shell) which is a more powerful and secure replacement for `rsh`. For more information on `ssh`, see `www.ssh.fi`[1].

[1] `http://www.ssh.fi`

If `DetachB` is **`false`**, a non-detached process is created. A non-detached process terminates as soon as the creating process does (think of crashes, there will be no orphaned processes). The lifetime of a detached process (that is, `DetachB` is **`true`**) is independent of the creating process.

On some platforms (in particular on `solaris-sparc`) the operating system in its default configuration does not support virtual sites efficiently. Namely, the Solaris OS limits the total number of shared memory pages per process to six and the number of shared memory pages system-wide to 100, while each connected Mozart process requires at least two shared memory pages for efficient communication. Please ask your system administrator to increase those limits with respect to your needs.

The Mozart system tries to do its best to reclaim shared memory identifiers, even upon process crashes, but it is still possible that some shared memory pages become unaccounted and thus stay forever in the OS. In these cases please use Unix utilities (on Solaris and Linux these are `ipcs` and `ipcrm`) to get rid of them.

`link`

```
link(url:+UrlV ModuleR <= _)
link(name:+NameV ModuleR <= _)
```

Links the module identified either by a url `UrlV` (a virtual string) or a module name `NameV` (a virtual string). Returns a module `ModuleR`.

For explanation see Chapter 2.

`apply`

```
apply(+Functor ModuleR <= _)
apply(url:+UrlV +Functor ModuleR <= _)
apply(name:+NameV +Functor ModuleR <= _)
```

Applies the functor `Functor`, where the url `UrlV` (a virtual string) or the module name `NameV` (a virtual string) serve as base url for resolving the functor's import.

For explanation see Chapter 2.

`enter`

```
enter(url:+UrlV ModuleR)
enter(name:+NameV ModuleR)
```

Installs the module `ModuleR` under the url `UrlV` (a virtual string) or the module name `NameV` (a virtual string).

For explanation see Chapter 2.

`ping`

```
ping()
```

Raises exception if remote process is dead. Blocks until executed by remote process.

`close`

```
close()
```

Performs a controlled shutdown of all remote processes (for a discussion of controlled shutdown see Section *Controlled system shutdown, (Distributed Programming in Mozart - A Tutorial Introduction)*).

12.1 Process Termination and Remote Managers

Here are some tentative explanations of what happens to the children of a process when the latter is terminated.

- if a process is properly shutdown, then detached children survive and non-detached children are terminated.
- if a process is killed with `kill -INT`, then its children are terminated whether they are detached or not.
- if a process is killed with `kill -KILL`, then no child is terminated because the proper shutdown sequence is not executed.
- if a process is killed by typing `C-c`, then the `INT` signal is sent to the *process group* to which both parent and children belong.[2] Thus all are terminated.

[2]This is the case in the current regime, but could be changed if desired

Referring To Distributed Entities: URL

In the age of the *World Wide Web*, resources needed by a running system don't just reside in files, they reside at URLs. The URL module provides an interface for creating and manipulating URLs as data-structures. It fully conforms to URI syntax as defined in RFC 2396 Uniform Resource Identifiers (URI): Generic Syntax[1] by T. Berners-Lee, R. Fielding, and L. Masinter (August 1998), and passes all 5 test suites published by Roy Fielding.

The only derogations to said specification were made to accommodate Windows-style filenames: (1) a prefix of the form *C*`:` where *C* is a single character is interpreted as Windows-style device notation rather than as a uri scheme – in practice, this is a compatible extension since there are no legal single character schemes, (2) path segments may indifferently be separated by `/` or \; this too is compatible since non-separator forward and backward slashes ought to be otherwise *escape encoded*.

There is additionally a further experimental extension: all urls may be suffixed by a string of the form "{foo=a,bar=b}". This adds an *info* record to the parsed representation of the url. Here, this record would be `info(foo:a bar:b)`. Thus properties can be attached to urls. For example, we may indicate that a url denotes a native functor thus: `file:/foo/bar/baz.so`{`native`}. Here `{native}` is equivalent to `{native=}`, i.e. the info record is `info(native:'')`.

13.1 Examples

Here are a few examples of conversions from url vstrings to url records. Return values are displayed following the function call.

```
{URL.make "http://www.mozart-oz.org/home-1.1.0/share/FD.ozf"}
```

```
url(
   absolute  : true
   authority : "www.mozart-oz.org"
   device    : unit
   fragment  : unit
   info      : unit
   path      : ["home-1.1.0" "share" "FD.ozf"]
   query     : unit
   scheme    : "http")
```

[1] `ftp://ftp.isi.edu/in-notes/rfc2396.txt`

The `absolute` feature has value `true` to indicate that the path is absolute i.e. began with a slash. The `path` feature is simply the list of path components, as strings.

`{URL.make "foo/bar/"}`

```
url(
   absolute  : false
   authority : unit
   device    : unit
   fragment  : unit
   info      : unit
   path      : ["foo" "bar" nil]
   query     : unit
   scheme    : unit)
```

The above illustrates a relative url: the `absolute` feature has value `false`. Note that the trailing slash results in the empty component `nil`.

`{URL.make "c:\\foo\\bar"}`

```
url(
   absolute  : true
   authority : unit
   device    : &c
   fragment  : unit
   info      : unit
   path      : ["foo" "bar"]
   query     : unit
   scheme    : unit)
```

Here the leading `c:` was parsed as a Windows-style device notation and the backslashes as component separators.

`{URL.make "foo.so{native}"}`

```
url(
   absolute  : false
   authority : unit
   device    : unit
   fragment  : unit
   info      : info(native:nil)
   path      : ["foo.so"]
   query     : unit
   scheme    : unit)
```

The `{native}` annotation is entered into the `info` feature.

13.2 Interface

`URL.make`

```
{URL.make +VR ?UrlR}
```

Parses virtual string VR as a url, according to the proposed uri syntax modulo Windows-motivated derogations (see above). Local filename syntax is a special case of schemeless uri. The parsed representation of a url is a non-empty record whose features hold the various parts of the url, it has the form `url(...)`. We speak of url records and url vstrings: the former being the parsed representation of the latter. A url record must be non-empty to distinguish it from the url vstring consisting of the atom `url`. The empty url record can be written e.g. `url(unit)`. VR may also be a url record, in which case it is simply returned.

`URL.is`

```
{URL.is +X}
```

Returns **`true`** iff X is a non-empty record labeled with `url`.

`URL.toVirtualString`

```
{URL.toVirtualString +VR ?V}
```

VR may be a url record or a virtual string. The corresponding normalized vstring representation is returned. `#FRAGMENT` and {`INFO`} segments are not included (see below). This is appropriate for retrieval since fragment and info sections are meant for client-side usage.

`URL.toVirtualStringExtended`

```
{URL.toVirtualStringExtended +VR +HowR ?V}
```

Similar to the above, but `HowR` is a record with optional boolean features `full`, `cache`, and `raw`. `full:true` indicates that *#FRAGMENT* and {*INFO*} sections should be included if present. `cache:true` requests that cache-style syntax be used (see Chapter 14): the `:` following the scheme and the `//` preceding the authority are both replaced by single `/`. `raw:true` indicates that no escape encoding should take place; this is useful e.g. for Windows filenames that may contain spaces or other characters illegal in URIs.

`URL.toString`

```
{URL.toString +VR ?S}
```

Calls `URL.toVirtualString` and converts the result to a string.

`URL.toAtom`

```
{URL.toAtom +VR ?A}
```

Calls `URL.toVirtualString` and converts the result to an atom.

`URL.resolve`

```
{URL.resolve +BaseVR +RelVR ?UrlR}
```

`BaseVR` and `RelVR` are url records or vstrings. `RelVR` is resolved relative to `BaseVR` and a new url record is returned with the appropriate fields filled in.

`URL.normalizePath`

```
{URL.normalizePath +Xs ?Ys}
```

Given a list `Xs` of string components (see `path` feature of a url record), returns a list `Ys` that results from normalizing `Xs`. Normalization is the process of eliminating occurrences of path components `.` and `..` by interpreting them relative to the stack of path components. A leading `.` is preserved because `./foo` and `foo` should be treated differently: the first one is an absolute path anchored in the current directory, whereas the second one is relative.

`URL.isAbsolute`
`URL.isRelative`

```
{URL.isAbsolute +VR ?B}
{URL.isRelative +VR ?B}
```

A url is considered absolute if (1) it has a `scheme`, or (2) it has a `device`, or (3) its `path` started with `/`, `~` (user home directory notation), `.` (current directory), or `..` (parent directory). For example, `~rob/foo/baz` is absolute.

`URL.toBase`

```
{URL.toBase +VR ?UrlR}
```

Turns a url vstring or record into a url record that can safely be used as a base for `URL.resolve` without loosing its last component. Basically, it makes sure that there is a slash at the end.

14

Resolving URLs: Resolve

The Resolve module generalizes the idea of a *search path* and simplifies read-oriented operations on arbitrary files and urls. The reader should be warned that this module has not yet reached full maturity.

14.1 From search paths to search methods

A search path is a list of directores sequentially searched to resolve a relative pathname. On Unix a search path is traditionally specified by an environment variable whose value is of the form:

```
Dir1:Dir2:...:DirN
```

On Windows, the colons : would be replaced by semi-colons ;. In the age of the *World Wide Web*, the classical notion of a search path is too limited: we want to search for arbitrary urls in arbitrary networked locations, and not simply for relative pathnames in local directories. For this reason, the notion of a directory to be searched is replaced by that of a method to be applied. A sequence of search methods can be specified by an environment variable whose value is of the form:

```
Meth1:Meth2:...:MethN
```

where each MethK is of the form KIND=ARG. KIND selects the method to be applied and ARG is its parameters. On Windows, the colons might be replaced by semi-colons, we support both notations on all platforms. The idea is of course that each method should be tried until one of them succeeds in locating the desired resource.

14.2 Syntax of methods

We now describe the syntax of KIND=ARG for the supported methods. For each one, we use a concrete example. ARG can normally be indifferently a directory or a url.

Any character of ARG can be escaped by preceding it with a backslash \: this is useful e.g. to prevent an occurrence of a colon in a url to be interpreted as a method separator. However, it means that, if you insist on using \ as a path component separator (à la Windows) instead of / (à la Unix), then you will have to escape them in ARG. Furthermore, \ is also an escape character for the shell, which means that you will normally have to double each escape character.

`all=/usr/local/oz/share`

The last component in the input url is extracted and looked up in the location supplied as the methods argument. If the input url is `http://www.mozart-oz.org/home/share/Foo.` then we try to look up `/usr/local/oz/share/Foo.ozf` instead.

`root=~/lib/oz`

This applies only to a relative pathname: it is resolved relative to the base url or directory supplied as argument to the method. If the input url is `share/Foo.ozf` then `~/lib/oz/share/Foo.ozf` is looked up instead. For convenience, and to be compatible with search path notation, you can omit `root=` and simply write this method as `~/lib/oz`

`cache=/usr/local/oz/cache`

Applies only to a full url: it is transformed into a relative pathname and looked up in the specified location. If the input url is: `http://www.mozart-oz.org/home/share/Foo.ozf`, then `/usr/local/oz/cache/http/www.mozart-oz.org/home/share/Foo.ozf` is looked up instead. This method is typically used to permit local caching of often used functors. The cache location could also be the url of some sort of mirroring server.

`prefix=http://www.mozart-oz.org/home/=~/oz/`

This method has the form `prefix=LOC1=LOC2`. Whenever the input url begins with the string `LOC1`, this prefix is replaced by `LOC2` and the result is looked for instead. Thus, if the input url is `http://www.mozart-oz.org/home/share/Foo.ozf`, we would look for `~/oz/share/Foo.ozf`.

`pattern=http://www.?{x}/home/?{y}=ftp://ftp.?{x}/oz/?{y}`

The `pattern` method is more general than the `prefix` method. `LOC1` can contain match variables, such as `?{x}` and `?{y}` that should match arbitrary sequences of characters, and `LOC2` can also mention these variables to denote the value they have been assigned by the match. Thus, if the input url is `http://www.mozart-oz.org/home/share/Foo.ozf`, we would look for `ftp://ftp.mozart-oz.org/oz/share/Foo.ozf`.

`=`

Normally, the default handler is implicitly appended to your search methods. This is the handler that simply looks up the input url itself, when all previous methods have failed. Sometimes it is desirable to disallow this default: for example this is the case when building the mozart distribution; the build process should be self contained and should not attempt to access resources over the network. You can disallow the default by appending `=` as the very last of your search methods. Thus

```
.:all=~/oz/bazar:=
```

would first try to resolve relative pathnames with respect to the current directory, then all urls by looking up their final component in directory `~/oz/bazar`, and that's it. If neither of these two methods succeeds, the resolution would simply raise an exception, but it would not attempt to retrieve the input url from the net.

14.3 Interface of `Resolve` Module

A resolver is a module that encapsulates and exports the resolving services of a sequence of search methods. For different purposes, you may need to apply different resolution strategies. For this reason, you may create arbitrarily many resolvers, each implementing an arbitrary resolution strategy.

`Resolve.make`

```
{Resolve.make +VS +Spec ?R}
```

Creates a new resolver `R`, identified by virtual string `VS` in trace messages, and whose strategy is initialized according to `Spec` which is one of:

`init(L)`

where `L` is a list of handlers (see later).

`env(V)`

where `V` names an environment variable whose value provides the search methods. If it is not set, the initial strategy simply looks up the input url itself.

`env(V S)`

same as above, but, if the environment variable is not set, then use `S` as its value.

`vs(S)`

simply get the search methods from virtual string `S`.

`Resolve.trace.get`
`Resolve.trace.set`

```
{Resolve.trace.get ?Bool}
{Resolve.trace.set +Bool}
```

Obtain or set the trace flag. When tracing is enabled, every resolve method that is attempted prints an informative message. Furthermore, all messages are prefixed by the virtual string identifying the resolver in which these methods are being invoked.

`Resolve.expand`

```
{Resolve.expand +Url1 ?Url2}
```

Takes a Url or virtual string as input and returns a Url with `"~"` expanded to the full user's home directory path, `"~john"` expanded to `john`'s home directory, `"."` and `".."` expanded to the current directory and parent directory. This functionality really belongs in the `URL` module, but is put here instead to keep module `URL` stateless.

`Resolve.handler`

You don't have to specify your methods as virtual strings, instead you can directly construct them using the following procedures:

`Resolve.handler.default`

This is the default handler that simply looks up the given url as is.

`Resolve.handler.all`

```
{Resolve.handler.all +LOC ?Handler}
```

This creates a handler that implements the `all` method for location `LOC`. The final component in the input url is looked up in `LOC`.

Resolve.handler.root

`{Resolve.handler.root +LOC ?Handler}`

This creates a handler that implements the `root` method for location `LOC`. A relative pathname is resolved relative to `LOC`.

Resolve.handler.cache

`{Resolve.handler.cache +LOC ?Handler}`

This creates a handler that implements the `cache` method for location `LOC`. A full url is transformed into a relative pathname and resolved relative to `LOC`.

Resolve.handler.prefix

`{Resolve.handler.prefix +Prefix +Subst ?Handler}`

This creates a handler that implements the `prefix` method. If the input url begins with string `Prefix`, then this is replaced by `Subst` and looked up instead.

Resolve.handler.pattern

`{Resolve.handler.prefix +Pattern +Subst ?Handler}`

This creates a handler that implements the `pattern` method. If the input url matches the string pattern `Pattern`, then this is replaced by the corresponding instantiation of `Subst` and looked up instead.

14.4 Interface of a Resolver

Each resolver `R` exports the following methods

R.getHandlers

`{R.getHandlers ?L}`

obtains `R`'s current list of handlers.

R.setHandlers

`{R.setHandlers +L}`

install's `L` as `R`'s current list of handlers.

R.addHandler

`{R.addHandler front(H)} {R.addHandler back(H)}`

adds `H` at the front (resp. at the back) of `R`'s list of handlers.

R.localize

`{R.localize +Url ?Rec}`

the return value `Rec` is `old(Filename)` if `Url` resolves to local file `Filename`, else it is `new(Filename)` where `Filename` is a new local file created by retrieving the data at `Url`.

R.open

`{R.open +Url ?FdI}`

returns `FdI`, which is an integer file descriptor open for read on the data available from `Url`.

R.load

`{R.load +Url ?V}`

returns the value `V` obtained from the pickle available at `Url`.

`R.native`

```
{R.native +Url ?M}
```

returns the native module `M` obtained by dynamically linking the native functor available in file `Url`.

15

Detecting and Handling Distribution Problems: `Fault`

This section summarizes the operations of the `Fault` module and their argument types. Please refer to the Distribution Tutorial for a full specification of the operations and examples of how to use them. This section carefully indicates where the current release is incomplete with respect to the specification (called a *limitation*) or has a different behavior (called a *modification*).

15.1 Argument Types

We summarize the argument types for the operations in the `Fault` module.

`Entity`

A reference to any Oz language entity that has distributed fault modes, namely any object, cell, lock, port, or logic variable.

`Level`

Either `site` or `'thread'(T)`, where `T` is a thread reference or the atom `this`.[1]

`FStates`

A set of fault states, i.e., a list that can contain at most one of each of the elements `tempFail`, `permFail`, `remoteProblem(tempSome)`, `remoteProblem(permSome)`, `remoteProblem(t`
and `remoteProblem(permAll)`.

`OP`

A record that indicates which attempted operation caused the exception or handler invocation. The value of `OP` is one of:

- `bind(T)`, `wait`, `isDet` (for logic variables).
- `cellExchange(Old New)`, `cellAssign(New)`, `cellAccess(Old)` (for cells).
- `'lock'` (for locks).
- `send(Msg)` (for ports).

[1]Since **`thread`** is already used as a keyword in the language, it has to be quoted to make it an atom.

- `objectExchange(Attr Old New)`, `objectAssign(Attr New)`, `objectAccess(Attr Old)`, `objectFetch` (for objects). A limitation of the current release is that an attempted operation on an object cannot be retried.

`HandlerProc`

A handler, i.e., a three-argument procedure that is called as `{HandlerProc Entity FStates OP}`, where `FStates` is a set of currently active fault states. A handler replaces an attempted operation on an entity.

`WatcherProc`

A watcher, i.e., a two-argument procedure that is called in its own thread as `{WatcherProc Entity FS` where `FStates` is a set of currently active fault states. A watcher is invoked as soon as the site detects a fault.

15.2 Fault Information

When there is a distribution problem, then three items of information are made available:

- `Entity`: the faulty entity.
- `ActualFStates`: the fault states that are currently active. This is always a subset of the states that the entity is set up to detect. For objects, cells, and locks, the fault states `tempFail(info:I)` and `permFail(info:I)` are possible, where `I` is in {`state`, `owner`}. This tells whether the fault is due to a lost state pointer (`state`) or a crashed owner (`owner`).
- `OP`: the operation that is attempted but does not succeed.

The system can be configured (see below) so that these three items appear in one or more of the following three ways:

- In an exception with format `system(dp(entity:Entity conditions:FStates op:OP) ...)`
- As arguments to a handler call, `{HandlerProc Entity FStates OP}`.
- As arguments to a watcher call, `{WatcherProc Entity FStates}`.

A limitation of the current release is that the `Entity` argument is undefined for an object operation. For handlers and watchers, this limitation can be bypassed by giving the handler and watcher procedures a reference to the object.

15.3 Operations

The `Fault` module contains the following operations. All operations return a boolean flag `B` that is **`true`** if the operation succeeds and **`false`** otherwise. All `enable` and `install` operations succeed if nothing was enabled or installed at that level. An entity with a successful `enable` or `install` at a given level is said to *have fault detection* at that level. All `disable` and `deInstall` operations succeed if nothing was disabled or deinstalled at that level. The system starts up as if `{Fault.defaultEnable [tempFail permFail] _` was executed.

All the following operations that have an `Entity` argument will do nothing if entity does not have distributed fault modes. If a logic variable with fault detection is bound to a nonvariable entity, then the fault detection is transferred to the entity, provided the latter has no fault detection at that level.

`{Fault.defaultEnable FStates ?B}`

Sets the default fault detection to `FStates` on the current site. When an operation is attempted on an entity and there is no fault detection on the site or thread level for the entity, then the default fault detection is used. This always succeeds.

`{Fault.defaultDisable ?B}`

Sets the default fault detection to `nil` on the current site. This always succeeds.

`{Fault.enable Entity Level FStates ?B}`

Enables fault detection on a given entity at a given level for a given set of fault states. An exception is raised if a fault is detected when an operation is attempted on the entity.

`{Fault.disable Entity Level ?B}`

Disables fault detection on a given entity at a given level.

`{Fault.install Entity Level FStates HandlerProc ?B}`

Installs a handler for fault detection on a given entity at a given level for a given set of fault states. The handler `{HandlerProc Entity AFStates OP}` is called if a fault is detected when an operation is attempted on the entity. A modification of the current release with respect to the specification is that handlers installed on variables always retry the operation after they return.

`{Fault.deInstall Entity Level ?B}`

Deinstalls a handler for fault detection on a given entity at a given level.

`{Fault.installWatcher Entity FStates WatcherProc ?B}`

Installs a watcher for fault detection on a given entity for a given set of fault states. Any number of watchers can be installed on an entity. It is always possible to install a watcher, so therefore this always succeeds. The watcher `{WatcherProc Entity AFStates}` is called in its own thread as soon as the site detects a fault.

`{Fault.deInstallWatcher Entity WatcherProc ?B}`

Deinstalls the given watcher on a given entity. This call succeeds if `WatcherProc` was installed on the entity. If there is more than one instance of `WatcherProc` installed on the entity, then exactly one is deinstalled.

On a given entity at the `global` level, at most one enable can be done or one handler installed. For a given entity, the `site` level can have at most one fault detection per site. The `'thread'(T)` can have at most one fault detection per thread. To have another fault detection, it is necessary to do a disable or deinstall first.

15.4 Limitations and Modifications

The current release has the following limitations and modifications with respect to the failure model specification. A *limitation* is an operation that is specified but not possible in the current release. A *modification* is an operation that is specified but behaves differently in the current release.

Most of the limitations and modifications listed here will be removed in future releases.

15.5 Limitations

The limitations are:

- The fault state `tempFail` is indicated only after a long delay. In future releases, the delay will be very short and based on adaptive observation of actual network behavior.
- If an exception is raised or a handler or watcher is invoked for an *object*, then the `Entity` argument is undefined. For handlers and watchers, this limitation can be bypassed by giving the handler and watcher procedures a reference to the object.
- If an exception is raised or a handler is invoked for an *object*, then the attempted object operation cannot be retried.

15.6 Modifications

The modifications are:

- A handler installed on a *variable* will retry the operation (i.e., bind or wait) after it returns. That is, the handler is inserted before the operation instead of replacing the operation.

16

Locating services in a network: Discovery

In order to make it easier to find a service (an Oz server application) one might want to have some sort of yellow pages, or a directory. While a directory is easy to implement in Oz, the `Discovery` module allows to locate it in a local area network.

The current implementation may be incomplete on certain platforms. Notably, only on Linux and Solaris it is guaranteed that broadcasts are sent to all the available networks. On other platforms this depends on the operating system.

The `Discovery` service consists of two parts, a server and a client. The server is initialized with a value. The server waits for inquiries from clients and sends the value as an answer. The client sends a broadcast message to all available networks (for example, ethernet and ip over serial link). Then the client waits for answers from servers.

A value the server holds would typically be a ticket to an Oz port that a directory server listens to.

16.1 The Module

The module has three features:

1. `server` The server class.
2. `client` The client class.
3. `defaultServerPort` The number of the default ip port that the server listens to.

The server class `Discovery.server` has following methods:

`init`

```
init(info:Info port:PortNr <= DefaultServerPort)
```

`Info` is the answer to be send on request by clients. It must be a virtual string. The server listens to the ip port `PortNr`. If the field `port` is not present, the default port number will be used.

replace

```
replace(info:Info)
```

Replaces the answer to be send to clients.

close

```
close()
```

Closes the operation of the server.

The client class `Discovery.client` has following methods:

init

```
init(port:ServerPortNr <= DefaultServerPort)
```

This method broadcasts a message. Answers to that message can be obtained using the methods `getOne` and `getAll`. The port that the server listens can be specified as `ServerPortNr`. If the field `port` is not present, the default port number will be used.

getOne

```
getOne(timeOut:TimeOut <= 1000 info:?Info)
```

There could be several servers listening to broadcasts from a client, therefore there can be several answers. If an answer is received before `TimeOut` milliseconds the variable `Info` will hold that answer. Otherwise `Info` will be `timeout`. Answers are Oz strings.

Instead of a time in milliseconds `TimeOut` can be `inf`. This means that the method will suspend until an answer is received (or forever if no answer is received).

If this method is called again the next answer (if such answer exists) will be returned.

getAll

```
getAll(timeOut:TimeOut <= 1000 info:?Info)
```

Method `getAll` can be used instead of method `getOne`. After calling this method, `Info` will hold a list of all answers received before `TimeOut` milliseconds has gone by.

This method is implemented using `getOne`. So any answers fetched by calling `getOne` will not reappear in the list `Info`, and vice versa.

`TimeOut` can be `inf` here too. In this case a stream will be returned instead of a list.

close

```
close()
```

Closes the operation of the client class.

17

Initializing and instrumenting the distribution layer: DPInit

The distribution layer of Mozart is dynamicly loaded when used. Some parameters can be defined at loading time. This section gives a brief description of the interface and what is possible to do with it.

17.1 Interface of DPInit Module

DPInit.init

`{DPInit.init +Spec ?B}`

Initializes the distribution layer of Mozart parameterized according to `Spec`. The distribution layer can only be initialized once, `DPInit.init` returns a boolean flag `B` that is true if the settings was accepted and false otherwise. The distribution layer can only be initialized once. `Spec` is an record of the following type:

```
init(ip:IpString <= SystemIp       port:IpPort <= 9000       acceptproc:AccProc <= DefaultAcc
```

All entries in the `Spec` record are optional, if `DPInit.init` is called with just the atom `init` as parameter the system will start with all values as default values. The fields and their implication on the system is defined here:

IpString

Defines the ip number that the Mozart site should expose as its home address to the outside world. Must be on the form `"193.10.66.192"`. The system will map the host name of the computer to an ip number if this field is not given. The Ip number should only be set if the operating system by some reason returns a faulty ip number.

IpPort

Sets the port number that should be used for listening on incomming connection atempts. If the choosen port number is unavailable a higher port number will be tired. The default value is 9000.

AccProc

Connection atempts comming in to the Site are handled by a piece of Mozart code called the AcceptProcedure. The AcceptProcedure defines a handshaking prototcol that the connecter must follow to be accepted. It can be replaced if other needs are present than what the default AcceptProcedure can give.

`ConProc`

Connection atempts done to other Sites are performed by a Mozart procedure called the ConnectionProcedure. The ConnectionProcedure used must be compatible with the othtre machine’s AcceptProcedure for the connection to be established. The ConnectionProcedure can be replaced with a custimzed ConnectionProcedure.

`FW`

The default schema for connection handeling is not realy suited for one-way firewalls. Sites sitiuated behind one-way firewalls should set the firewall flagg to true. If the firewall flag is true the Site will be reluctant to close down its connection due to resource shortage.

`DPInit.getSettings`

```
{DPInit.getSettings ?AnsSpec}
```

Returns the settings of the distribution layer. If the distribution layer was initialized a record of the same format as `Spec` is returned otherwise the atom `not_initilaized` is returned.

18

Retriving statistical information from the Distribution layer: `DPStatistics`

To be able to understand and tune distributed Mozart programs the distribution layer can produce log files and exposes interfaces to it's inner state.

18.1 Interface of `DPStatistics` Module

`DPStatistics.siteStatistics`

`{DPStatistics.siteStatistics ?SiteList}`

Returns a list of all currently know remote sites. `SiteList` contains records that has the following fields:

`ip`

The ip number of the remote site in string format.

`lastRTT`

The last measured round trip to the remote site in ms. If no connection is established the value is `~1`.

`pid`

The proces identifier of the remote sites proces.

`port`

The tcp port of the remote site. It is used for accepting connection atempts.

`received`

Number of received messages from the remote site. This number will be cleared at each invokation to `DPStatistics.siteStatistics`

`sent`

Number of messages sent to the remote site from this site. This number will be cleared at each invokation to `DPStatistics.siteStatistics`

`state`

The state of the remote site from the current sites point of view.

`table`

Internal information.

`siteid`

The unique string identifying the remote site.

`timestamp`

The time when the remote site was created.

`DPStatistics.getTablesInfo`

```
{DPStatistics.getTablesInfo ?TableInfo }
```

Returns information about the tables holding imported and exported entities. `TableInfo` is a list containing of the following format:

```
[bt(list:BTlist size:BTsize) ot(list:OTlist size:OTsize)]
```

`BTlist`

A list of all currently imported entities, each entry in the list are of the following format:

```
be(index:OTindex na:NetAddress primCred:PrimCredit secCred:SecCredit type:Type)
```

`BTsize`

The size of the Borrow table. The borrow table will grow and shrink to fit the number of imported entities.

`OTlist`

A list of all currently exported entities, each entry in the list are of the following format:

```
 oe(credit:PrimCredit index:OTindex type:Type)
```

`OTsize`

The size of the Owner table. The owner table will grow and shrink to fit the number of exported entities.

`DPStatistics.getNetInfo`

```
{DPStatistics.getNetInfo  ?NetInfo }
```

Returns information about internaly allocated object in the distribution layer. `NetInfo` is a list containing records with the following fields

`type`

The type of the object

`nr`

The number of allocated objects of this type

`size`

The size of this object in bytes.

`DPStatistics.perdioStatistics`

```
{DPStatistics.perdioStatistics   ?Ans }
```

The distribution layer keeps information of the number of received and sent messages per message type basis as the number of marshaled and unmarshaled marshaling types. This information is returned in record with the following fields:

`recv`

Contains a record with the total number of unmarshaled dif's per type and received messages per message type.

`sent`

Contains a record with the total number of marshaled dif's per type and sent messages per message type.

`DPStatistics.createLogFile`

`{DPStatistics.createLogFile +File}`

Directs the loging output from the distribution layer to the file `File`.

Part IV

Open Programming

Files, Sockets, and Pipes: Open

This chapter gives reference documentation for the Open module. The module contains the following classes:

1. Open.file for reading and writing files.
2. Open.socket for Internet socket connections.
3. Open.pipe for creation of operating system processes.
4. Open.text for reading and writing text line by line and character by character. It is a mixin class that can be combined with any of the classes of the Open module.

A tutorial account on open programming can be found in *"Open Programming in Mozart"*.

19.1 Exceptions

The methods of any of the Open module classes can raise different exceptions.

operating system When an operating system exception occurs the Oz exception defined in the module OS is raised.

already initialized An exception of the format

```
system(open(alreadyInitialized O M) debug:X)
```

is raised if an already initialized object O is initialized again by applying it to the message M.

already closed An exception of the format

```
system(open(alreadyClosed O M) debug:X)
```

is raised if a method other than close of an already closed object O is applied. M is as above the message the object has been applied to.

illegal flags An exception of the format

```
system(open(illegalFlags O M) debug:X)
```

is raised if an object `O` is initialized with an unknown flag. `M` is the initialization message.

ilegal modes An exception of the format

```
system(open(illegalModes O M) debug:X)
```

is raised if an object `O` is initialized with an unknown mode. `M` is the initialization message.

name or URL An exception of the format

```
system(open(nameOrUrl O M) debug:X)
```

is raised if an object `O` is initialized with both a name and an URL or with neither. `M` is the initialization message.

URL is read-only An exception of the format

```
system(open(urlIsReadOnly O M) debug:X)
```

is raised if an object `O` is initialized with an URL and the `write` flag. `M` is the initialization message.

19.2 The Class `Open.file`

The class `Open.file` has the following public methods.

`init`

```
init(name: +NameV
     flags:+FlagsAs <= [read]
     mode: +ModeR   <= mode(owner:[write] all:[read]))
init(url:  +UrlV
     flags:+FlagsAs <= [read]
     mode: +ModeR   <= mode(owner:[write] all:[read]))
```

Initializes the file object and associates it with a Unix file.

`NameV` is either a valid filename or one of the atoms `stdin`, `stdout`, and `stderr`. In this case, the standard input, standard output, or standard error stream is opened, respectively.

In addition to using a filename to open a file also a url `UrlV` can be used. Only one of the features `name` or `url` is allowed.

The value of `FlagsAs` must be a list, with its elements chosen from the following atoms:

`read`, `write`, `append`, `create`, `truncate`, `exclude`, `text`, `binary`

For reading a file, the atom `read` must be included in `FlagsAs`. Similarly, the atom `write` must be included for writing. It is possible to include both atoms, giving both read and write access to the file. For files attached to an url, only reading access is possible.

Atom `text` opens a file in text mode. This is important on platforms (e.g. Windows) where the line ends are represented on disk by `CRLF` rather than just `LF`. On such platforms, when a file is opened in text mode, the conversion from `CRLF` to `LF` on reading (and in the reverse direction on writing) happens automatically. Atom `binary` opens a file in binary mode, where no such translation happens. On other platforms (e.g. Linux), these flags have no effect.

When a file object is opened, the seek pointer, pointing to the current position in the file, is initialized to point to the start of the file. Any subsequent read or write takes place at the position given by this pointer.

The remaining atoms make sense only if the file is opened for writing. If the atom `append` is included, the seek pointer is moved to the end of the file prior to each attempt to write to the file.

If the file to be opened already exists, the presence of the atom `create` has no effect. Otherwise, the file is created. Including `truncate` resets the length of an existing file to zero and discards its previous content.

An attempt to open an existing file fails, if `exclude` is contained. Thus, this flag grants exclusive access of the file object to the operating system file.

If the file is opened for writing and the atom `create` is included, the access rights are set as specified by `ModeR`. This must be a record with fields drawn from `all`, `owner`, `group` and `others`. Its subterms must be lists of the atoms `read`, `write` and `execute`.

More detailed information can be found in `open(2)`, `chmod(2)`, and `umask(2)`.

`read`

```
read(list: ?ListS
     tail: TailX   <= nil
     size: +SizeAI <= 1024
     len:  ?LenI   <= _)
```

Reads data from a file. `SizeAI` specifies how much data should be read from the file. If the field `len` is present, `LenI` is bound to the number of bytes actually read. `LenI` may be less than `SizeAI`. The atom `all` is also a legal value for `SizeAI`. In this case the entire file is read.

The data read binds `ListS` to a list of characters. The tail of the list can be given by `TailX`. The value for `TailX` defaults to `nil`, which means that in this case the list `ListS` is a string.

See also `read(2)`.

`write`

```
write(vs:  +V
      len: ?I <= _)
```

Writes the virtual string V to a file.

See `write(2)`.

seek

```
seek(whence: +WhenceA <= set
     offset: +OffsetI <= 0)
```

Sets the file object's seek pointer

Allowed values for `WhenceA` are the atoms `set`, `current`, or `'end'`.

In case of `set` the position of the seek pointer is moved to the absolute position from the beginning of the file given by the value of `OffsetI`.

In case of `current` the pointer is moved ahead by `OffsetI`. Notice, that the pointer can be moved backward by a negative `OffsetI`, and forward by a positive `OffsetI`.

If `'end'` is given, the pointer is moved by `OffsetI` with respect to the current end of the file.

In particular, invoking `seek` with the default parameters moves the pointer to the beginning of the file.

See `lseek(2)`.

tell

```
tell(offset: ?OffsetI)
```

Returns the current position of the seek pointer counting from the beginning of the file.

See `lseek(2)`.

close

```
close
```

Closes the file object as well as the file.

See `close(2)`.

dOpen

```
dOpen(+ReadFileDescI +WriteFileDescI)
```

Initializes the object. `ReadFileDescI` and `WriteFileDescI` must be integers of already open file descriptors (in the usual operating system sense). Note that this method should only be used for advanced purposes.

getDesc

```
getDesc(?ReadFileDescIB ?WriteFileDescIB)
```

Returns the internally used file descriptors.

If the object is not yet initialized, `ReadFileDescIB` and `WriteFileDescIB` are bound to **`false`**, otherwise to the respective integers. Note, that this method is only for advanced purposes.

19.3 The Class Open.socket

The class Open.socket has the following public methods.

init

```
init(type:     +TypeA  <= stream
     protocol: +ProtoV <= ""
     time:     +TimeI  <= ~1)
```

Initializes a socket object.

The type of the socket TypeA determines the type of the socket, which can be either stream or datagram.

The protocol is described by ProtoV where the empty string "" means to choose an appropriate protocol automatically. Other possible values are the TCP protocol (you have to give "tcp") for stream sockets, and UDP (you have to give "udp") for datagram sockets.

The integer TimeI specifys for how long a time (in milliseconds) the socket attempts to accept a connection. The value ~1 means infinite time. See the following description of the accept method for more details.

See also socket(2).

bind

```
bind(takePort: +TakePortI <= _
     port:     ?PortI     <= _)
```

Names a socket globally.

If the field takePort is present, its value is chosen for binding. Otherwise, a fresh port number value is generated by the object. This port number is accessible at the field port.

See also bind(2).

listen

```
listen(backLog: +LogI <= 5)
```

Signals that a socket is willing to accept connections.

LogI describes the maximum number of pending connections to be buffered by the system.

See also listen(2).

accept

```
accept(accepted:    ?Object  <= _
       acceptClass: +Class   <= _
       host:        ?HostSB  <= _
       port:        ?PortIB  <= _)
```

Accepts a connection from another socket.

The method suspends until a connection has been accepted or the number of milliseconds as specified by the time value in the init method has elapsed. After this period, no connection will be accepted, and both PortIB and HostSB are bound to false.

If a connection is accepted within the given time, the following happens: `HostSB` and `PortIB` are bound accordingly if their fields are present.

If the fields `accepted` and `acceptClass` are present, `Object` is bound to an object created from the class `Class`. `Class` must be a sub class of `Open.socket`. Then the accepted connection is available with `Object`.

Otherwise, the access to the socket at which the connection was accepted, because any subsequent message will refer to the accepted socket connection.

See also `accept(2)`.

`connect`

```
connect(host: +HostV <= localhost
        port: +PortI)
```

Connects to another socket.

The address of the socket to connect to is given by `HostV` and `PortI`.

Be very careful in using this method: it blocks the entire Oz system until it succeeds.

See `connect(2)`.

`server`

```
server(port: ?PortI
       host: ?HostV <= localhost)
```

Initializes a stream socket as a server.

`client`

```
client(port: +PortI
       host: +HostV <= localhost)
```

Initializes a stream socket as a client.

`read`

```
read(list: ?ListS
     tail: TailX  <= nil
     size: +SizeAI <= 1024
     len:  ?LenI  <= _)
```

Receives data from a stream-connected socket or from a datagram socket with peer specified.

An attempt is made to read `SizeAI` bytes from the socket. `ListS` is constrained to the data while the tail of `ListS` is constrained to `TailX`. The atom `all` is also a legal value for `SizeAI`. In this case the entire input is read.

`LenI` is bound to the number of bytes actually read. If the socket is of type stream and the other end of the connection has been closed `LenI` is bound to `0`.

See also `read(2)`.

`receive`

```
receive(list: ?ListS
        tail: TailX  <= nil
        len:  ?LenI  <= _
        size: +SizeI <= 1024
        host: ?HostS <= _
        port: ?PortI <= _)
```

Receives data from a socket.

An attempt is made to read SizeI bytes from the socket. ListS is bound to the data while the tail of the list is bound to TailX.

LenI is bound to the number of bytes actually read. If the socket is of type stream and the other end of the connection has been closed LenI is bound to 0.

The source of the data is signaled by binding HostS and PortI.

See also recvfrom(2).

write

```
write(vs:  +V
      len: ?I <= _)
```

Writes the virtual string V to a stream-connected socket or to a datagram socket with peer specified.

I is bound to the number of characters written.

See also write(2).

send

```
send(vs:  +V
     len: ?I <= _)
send(vs:   +V
     len:  ?I     <= _
     port: +PortI
     host: +HostV <= localhost)
```

Sends data as specified by V to a socket.

The destination of the data may be given by HostV and PortI. If they are omitted, the data is sent to the peer of a datagram socket or to the other end of a connection in case of a stream socket. I is bound to the number of characters written.

See also send(2).

shutDown

```
shutDown(how: +HowAs <= [receive send])
```

Disallows further actions on the socket.

HowAs has to be a non-empty list which must contain only the atoms receive and send. The atom send signals that no further data transmission is allowed, while receive signals that no further data reception is allowed.

See also shutdown(2).

close

```
close
```

Closes the socket.

See also close(2)

flush

```
flush(how: +HowAs <= [receive send])
```

Blocks until all requests for reading, receiving, writing, and sending have been fulfilled.

HowAs must be a non-empty list which may include the atoms receive and send. The atom send signals that the method should block until all send (or write) requests are fulfilled, while receive signals the same for receive (or read).

dOpen

```
dOpen(+ReadFileDescI +WriteFileDescI)
```

Initializes the object. ReadFileDescI and WriteFileDescI must be integers of already open file descriptors (in the usual operating system sense).

Note that this method should only be used for advanced purposes.

getDesc

```
getDesc(?ReadFileDescIB ?WriteFileDescIB)
```

Returns the internally used file descriptors.

If the object is not yet initialized, ReadFileDescIB and WriteFileDescIB are bound to **false**, otherwise to the respective integers.

Note, that this method is only for advanced purposes.

19.4 The Class Open.pipe

The class Open.pipe has the following public methods.

init

```
init(cmd:  +CmdV
     args: +ArgsVs <= nil
     pid:  ?PidI   <= _)
```

Initilizes the object and forks a process with process identification PidI executing the command CmdV with arguments ArgsVs.

The environment of the forked process is inherited from the process which runs the Oz Emulator. The standard input of the forked process is connected to sending and writing data, the standard output and standard error to reading and receiving data.

See also execv(3), fork(2).

read

```
read(list: ?ListS
     tail: TailX  <= nil
     size: +SizeAI <= 1024
     len:  ?LenI  <= _)
```

Reads data *ListS* from the standard output or standard error of the forked process.

An attempt is made to read SizeI bytes. ListS is bound to the data read while the tail of ListS is bound to TailX. The atom all is also a legal value for SizeAI. In this case the entire input is read.

LenI is bound to the number of bytes actually read. If the socket is of type stream and the other end of the connection has been closed LenI is bound to 0.

See also read(2).

write

```
write(vs:  +V
      len: ?I <= _)
```

Writes the virtual string V to the standard input of the forked process.

I is bound to the number of characters written.

See also write(2).

flush

```
flush(how: +HowAs <= [receive send])
```

Blocks until all requests for reading and writing have been performed.

HowAs must be a non-empty list which may include the atoms receive and send. The atom send signals that the method should block until all write requests are fulfilled, while receive signals the same for read.

close

```
close(+KillB<=false)
```

Closes the object.

If KillB is **false** (the default) the method blocks until all pending read and write requests have been executed. If the started process is still running, it is killed by sending the SIGTERM signal. However, note that the inverse direction is not supported, which means the object is not automatically closed if the process terminates.

If KillB is **true** the possibly running process is immediately terminated by sending the SIGKILL signal.

See also wait(2) and kill(1).

dOpen

```
dOpen(+ReadFileDescI +WriteFileDescI)
```

Initializes the object. ReadFileDescI and WriteFileDescI must be integers of already open file descriptors (in the usual operating system sense).

Note that this method should only be used for advanced purposes.

getDesc

```
getDesc(?ReadFileDescIB ?WriteFileDescIB)
```

Returns the internally used file descriptors.

If the object is not yet initialized, ReadFileDescIB and WriteFileDescIB are bound to **false**, otherwise to the respective integers.

Note that this method is only for advanced purposes.

19.5 The Class `Open.text`

The mixin class `Open.text` has the methods listed below. Note, however, that it does not automatically cause a file to be opened in `text` mode; that must be done explicitly in the `init` method by supplying `text` as one of the flags.

`getC`

```
getC(?I)
```

Returns the next character, or **`false`** if the input is at the end.

Note that if an object is created that inherits from both `Open.text` and `Open.file`, the methods `read` and `seek` from the classes `Open.file`, `Open.socket`, and `Open.pipe` and do not work together with this method.

`putC`

```
putC(+I)
```

Writes the character `I`.

`unGetC`

```
unGetC
```

The last character read is written back to the input buffer and may be used again by `getC`. It is allowed only to unget one character.

`getS`

```
getS(?SB)
```

Returns the next line of the input as string, or **`false`** if the input is at the end. `SB` does not contain the newline character.

Note that if an object is created that inherits from both `Open.text` and `Open.file`, the methods `read` and `seek` from the class `Open.file`, `Open.socket`, and `Open.pipe` do not work together with this method.

`putS`

```
putS(+V)
```

Writes the virtual string `V`. Note that a newline character is appended.

`atEnd`

```
atEnd(?B)
```

Tests whether the end of input is reached.

`dOpen`

```
dOpen(+ReadFileDescI +WriteFileDescI)
```

Initializes the object. `ReadFileDescI` and `WriteFileDescI` must be integers of already open file descriptors (in the usual operating system sense).

Note that this method should only be used for advanced purposes.

`getDesc`

```
getDesc(?ReadFileDescIB ?WriteFileDescIB)
```

Returns the internally used file descriptors.

If the object is not yet initialized, `ReadFileDescIB` and `WriteFileDescIB` are bound to **`false`**, otherwise to the respective integers.

Note that this method is only for advanced purposes.

20

Operating System Support: OS

This chapter describes the procedures contained in the module OS

20.1 Conventions

Most procedures can be seen as straightforward lifting of POSIX compatible operating system functions to Oz. Hence, our description consists mainly of a reference to the relevant Unix manual page. If you are running on a Windows based platform you should consult a POSIX documentation (e.g. [3]).

A major convention is that most `int` arguments in C are integers in Oz. Moreover `char*` arguments are virtual strings, if they are used as input in C. If they are used as output in C they are strings. An *n*-ary C-function returning a value is implemented as a $n+1$-ary Oz procedure where the last argument serves as output position.

Whenever one needs predefined POSIX constants, they can be used as Oz atoms. Wherever bitwise disjunction of predefined constants is used in C, a list of atoms (the constant names) is allowed, which is interpreted as bitwise disjunction of their respective values.

The functionality of the module `OS` can be classified as procedures which are useful in applications, or procedures needed for building high level functionality. The latter are only interesting for programmers who want to build abstractions similar to those provided by the `Open` module.

20.2 Exceptions

If the operating system returns an error, an Oz Exception is raised which looks as follows:

```
system(os(A S1 I S2) debug:X)
```

where:

1. The atom `A` gives the category of the error (e.g. `os` for operating system or `net` for network layer).

2. The string `S1` describes the operating system routine that raised the exception.

3. The integer `I` gives an operating system dependent error number.
4. The string `S2` describes the exception by some text.
5. The value `X` might contain some additional debug information.

20.3 Random Integers

rand

`{OS.rand ?I}`

Returns a randomly generated integer.

See `rand(3)`.

srand

`{OS.srand +I}`

Sets the seed for the random number generator used by `OS.rand`. If `I` is `0`, the seed will be generated from the current time.

See `srand(3)`.

randLimits

`{OS.randLimits ?MinI ?MaxI}`

Binds `MinI` and `MaxI` to the smallest and greatest possible random number obtainable by `OS.rand`.

See `rand(3)`.

20.4 Files

tmpnam

`{OS.tmpnam ?FileNameS}`

Returns a freshly created full pathname.

See `tmpnam(2)`.

unlink

`{OS.unlink +PathV}`

Removes the file with name `PathV`.

See `link(2)`.

20.5 Directories

getDir

`{OS.getDir +PathV ?FileNameSs}`

Returns a list of strings giving the files in the directory `PathV`.

See `opendir(3)` and `readdir(3)`.

getCWD

```
{OS.getCWD ?FileNameS}
```

Returns in `FileNameS` the path of the current working directory.

See `getcwd(3)`.

`stat`

```
{OS.stat +PathV ?StatR}
```

Returns a record describing the status of the file `PathV`.

`StatR` has features `size`, `type`, and `mtime`. The subtree at the feature `size` is an integer that gives the size of the file `PathV`. The subtree at the feature `type` is one of the following atoms: `reg` (regular file), `dir` (directory), `chr` (character special file), `blk` (block special file), `fifo` (pipe or FIFO special file), `unknown` (something else). `mtime` is the time of last modification measured in seconds since the dawn of the world, which, as we all know, was January 1, 1970.

See `stat(2)`.

`chDir`

```
{OS.chDir +PathV}
```

Changes the current working directory to `PathV`.

See `chdir(2)`.

`mkDir`

```
{OS.mkDir +PathV +ModeAs}
```

Creates a directory `PathV`. Access modes are specified by `ModeAs`, similar to `OS.open`.

See `mkdir(2)`.

20.6 Sockets

`getServByName`

```
{OS.getServByName +NameV +ProtoV ?PortIB}
```

Returns the port number `PortI` of a service `NameV` reachable in the Internet domain with the specified protocol `ProtoV`.

If the service is unknown, **`false`** is returned.

As an example, the application

```
{OS.getServByName "finger" "tcp" Port}
```

binds `Port` to the number, where you can connect to your local finger server.

See `getservbyname(3)`.

`getHostByName`

```
{OS.getHostByName +NameV ?HostentR}
```

Returns name information for the host `NameV`.

The record *HostentR* has the following features:

`name`	official host-name	string
`aliases`	alternative host-names	list of strings
`addrList`	Internet addresses	list of strings

See `gethostbyname(3)`.

20.7 Time

`time`

```
{OS.time ?TimeI}
```

Returns the time since 00:00:00 GMT, Jan. 1, 1970 in seconds.

See `time(2)`.

`gmTime`

```
{OS.gmTime ?GmTimeR}
```

`localTime`

```
{OS.localTime ?LocalTimeR}
```

Returns a description of the Coordinated Universal Time (UTC), respectively a description of the local time.

The records `GmTimeR` and `LocalTimeR` have the following features, where the fields are all integers:

`sec`	seconds	0– 61
`min`	minutes	0– 59
`hour`	hours	0– 23
`mDay`	day of month	1– 31
`mon`	month of year	0– 11
`year`	years since 1900	
`wDay`	days since Sunday	0– 6
`yDay`	day of year	0– 365
`isDst`	1 if daylight savings time in effect (DST)	

See `gmtime(3)`, `localtime(3)`, and `time(2)`.

20.8 Environment Variables

`getEnv`

```
{OS.getEnv +NameV ?ValueSB}
```

Returns the value of the environment variable `NameV`.

If a variable with the given name does not exist, the procedure returns **`false`**.

As an example, consider:

```
{OS.getEnv 'OZHOME'}
```

returns where the Oz system has been installed. This information is also available via `{System.get home}`

See `getenv(3)`.

`putEnv`

```
{OS.putEnv +NameV +ValueV}
```

Sets the value of the environment variable `NameV` to `ValueV`.

See `putenv(3)`.

20.9 Miscellaneous

`uName`

```
{OS.uName ?UtsnameR}
```

Returns system information.

The record `UtsnameR` has at least the following features, where all fields are strings:

`sysname`	operating system name
`nodename`	computer name
`release`	operating system release
`version`	operating system version
`machine`	machine architecture

See `uname(2)`.

`system`

```
{OS.system +CmdV ?StatusI}
```

Starts a new operating system shell in which the OS command `CmdV` is executed. The status of the command is reported by `StatusI`.

See `system(3)`.

20.10 Low Level Procedures

20.10.1 Basic Input and Output

`open`

```
{OS.open +FileNameV +FlagsAs +ModeAs ?DescI}
```

Opens a file for reading and/or writing.

`FlagsAs` must be a list with some of the following atoms as elements:

'O_RDONLY'	'O_WRONLY'	'O_RDWR'
'O_APPEND'	'O_CREAT'	'O_EXCL'
'O_TRUNC'	'O_NOCCTY'	'O_NONBLOCK'
'O_SYNC'		

Their meanings coincide with their usual POSIX meanings.

In the same manner `ModeAs` must be a list with elements drawn from:

'S_IRUSR'	'S_IWUSR'	'S_IXUSR'
'S_IRGRP'	'S_IWGRP'	'S_IXGRP'
'S_IROTH'	'S_IWOTH'	'S_IXOTH'

See `open(2)` and `chmod(2)`.

fileDesc

```
{OS.fileDesc +FileDescA ?FileDescIB}
```

Maps the atoms

```
'STDIN_FILENO'
'STDOUT_FILENO'
'STDERR_FILENO'
```

to integers, giving their respective file descriptor. Note that these descriptors will be duplicated.

See `open(2)` and `dup(2)`.

read

```
{OS.read +DescI +MaxI ?ListS TailX ?ReadI}
```

Reads data from a file or socket. Yields a list of characters in `ListS`, where the tail of the list is constrained to `TailX`.

Implements the `read(2)` system call.

write

```
{OS.write +DescI +V ?StatusTI}
```

Writes the virtual string `V` to a file or a socket by using the function `write(2)`.

Illegal parts of the virtual string `V` are simply ignored, and the legal parts are written.

If `V` contains an undetermined variable, `StatusTI` is bound to a ternary tuple with label `suspend`. The first argument is an integer describing the portion already written, the second the undetermined variable, and the last to the not yet written part of `V`.

Otherwise `StatusTI` is bound to the number of characters written.

lSeek

```
{OS.lSeek +DescI +WhenceA +OffsetI ?WhereI}
```

Positions the seek pointer of a file. This procedure is implemented by the `lseek(2)` function. `WhenceA` must be one of the atoms `'SEEK_SET'`, `'SEEK_CUR'`, and `'SEEK_END'`.

close

```
{OS.close +DescI}
```

Closes a file or socket by using the `close(2)` function.

20.10.2 From Blocking to Suspension

If reading from a file or a socket via an operating system function, it is possible that no information is available for reading. Furthermore, an attempt to write something to a file or to a socket might not be possible at a certain point in time. In this case the OS function blocks, i.e. the whole Oz Emulator process will stop doing any work.

To overcome this problem, we provide three procedures. These procedures will suspend rather than block.

acceptSelect

`{OS.acceptSelect +DescI}`

readSelect

`{OS.readSelect +DescI}`

writeSelect

`{OS.writeSelect +DescI}`

Blocks until a socket connection can be accepted (data is present to be read, and writing of data is possible, respectively) at the file or socket with descriptor `DescI`

For example the following case statement (we assume that `Desc` is bound to a descriptor of a socket):

```
{OS.readSelect Desc}
case {OS.read Desc 1024 ?S nil}
of ... then...
end
```

blocks the current thread until data for reading is available.

This functionality is implemented by the `select(2)` system call.

Before actually closing a file descriptor, the following procedure needs to be called.

deSelect

`{OS.deSelect +DescI}`

Discards all threads depending on the file descriptor `DescI`.

20.10.3 Sockets

socket

`{OS.socket +DomainA +TypeA +ProtoV ?DescI}`

Creates a socket. Is implemented by the function `socket(2)`.

`DomainA` must be either the atom `'PF_INET'` or `'PF_UNIX'`, whereas `TypeA` must be either `'SOCK_STREAM'`, or `'SOCK_DGRAM'`.

`ProtoV` must be a virtual string. If it denotes the empty string, an appropriate protocol is chosen automatically, otherwise it must denote a valid protocol name like `"tcp"` or `"udp"`.

bind

`{OS.bind +SockI +PortI}`

Binds a socket to its global name.

See `bind(2)`.

listen

`{OS.listen +SockI +BackLogI}`

Indicates that a socket is willing to receive connections.

See `listen(2)`.

accept

`{OS.accept +SockI ?HostS ?PortI ?DescI}`

Accepts a connect request on a socket.

`HostS` is bound to a string describing the host name. It is possible to use `OS.acceptSelect` to block a thread until a connect attempt on this socket is made.

See `accept(2)` and `gethostbyaddr(3)`.

`connect`

`{OS.connect +SockI +HostV +PortI}`

Connects to a socket.

See `connect(2)` and `gethostbyaddr(3)`.

`shutDown`

`{OS.shutDown +SockI +HowI}`

Signals that a socket is not longer interested in sending or receiving data.

See `shutdown(2)`.

`getSockName`

`{OS.getSockName +SockI ?PortI}`

Gets the name of a socket.

See `getsockname(3)`.

`send`

`{OS.send +SockI +MsgV +FlagsAs ?LenI}`

`sendTo`

`{OS.sendTo +SockI +MsgV +FlagsAs +HostV +PortI} ?LenI}`

Sends data from a socket.

`FlagsAs` must be a list of atoms; its elements must be either `'MSG_OOB'` or `'MSG_DONTROUTE'`.

See `send(2)` and `gethostbyname(3)`.

`receiveFrom`

```
{OS.receiveFrom +SockI +MaxI +FlagsAs ?MsgS
                +TailX ?HostS ?PortI ?LenI}
```

Receives data at a socket.

`FlagsAs` must be a list of atoms; its elements must be either `'MSG_OOB'` or `'MSG_PEEK'`.

See `recvfrom(2)` and `gethostbyaddr(3)`.

20.10.4 Process Control

`kill`

`{OS.kill +PidI +SigA ?StatusI}`

Implements the `kill(2)` function. `PidI` is the process identifier, `SigA` is an atom describing the signal to be sent: depending on the platform different signals are supported; at least the following signals are supported on all platforms: `'SIGTERM'`, `'SIGINT'`.

`pipe`

`{OS.pipe +CmdV +ArgsVs ?PidI ?StatusT}`

Forks a OS process which executes the command `CmdV` with arguments `ArgsVs`.

`PidI` is bound to the process identifier. `StatusT` is bound to a pair of socket or file descriptors where the standard input, the standard output and the standard error is redirected to. The first field of the pair is the descriptor for reading whereas the second field is the descriptor for writing.

See `socketpair(2)`, `fork(2)`, `execvp(2)`, and `execve(2)`.

`wait`

`{OS.wait ?PidI ?StatI}`

Implements the `wait(2)` function.

`getPID`

`{OS.getPID ?PidI}`

Returns the current process identifier as integer. See `getpid(2)`.

Part V

System Programming

Persistent Values: `Pickle`

The `Pickle` module provides procedures to store and retrieve stateless values on persistent storage.

`save`

```
{Pickle.save +X +PathV}
```

Stores `X` in a file named `PathV`.

Note that `X` can be any *stateless* value. So it is possible to save for example records, procedures or classes. However an exception is raised if for example `X` contains an object or a logic variable.

`saveCompressed`

```
{Pickle.saveCompressed X +PathV +LevelI}
```

Works like `save` but additionally compresses its output. `LevelI` is an integer between `0` and `9` specifying the compression level: the higher the value the better the compression factor, but the longer compression takes. A value of `0` gives no compression, so `{Pickle.save X Value}` is equivalent to `{Pickle.saveCompressed X Value 0}`.

Compression time and ratio depend on the type of input. The compression ratio might vary between 20 and 80 percent, while compression at level 9 is usually less than 2 times slower than using no compression.

`saveWithHeader`

```
{Pickle.saveWithHeader X +PathV +HeaderV +LevelI}
```

This procedure is a generalization of the above builtins. It saves `X` in file +`PathV` with compression level +`LevelI` and additionally prepends the virtual string `HeaderV` at the beginning. So `HeaderV` can be used for example to prepend a comment in front of the pickle or to prepend a shell startup script to load and execute the pickle.

`loadWithHeader`

```
{Pickle.loadWithHeader +UrlV ?Pair}
```

This procedure retrieves a value from URL `UrlV` that has been previously saved with one of the above procedures. It returns a pair `HeaderV#Value`, where `HeaderV` is the (possibly empty) header and `Value` the value that was retrieved.

`load`

```
{Pickle.load +UrlV ?Value}
```

This is just a shortcut for

```
{Pickle.loadWithHeader UrlV _#Value}
```

pack

```
{Pickle.pack +X ?ByteString}
```

Takes a value X and pickles it to a bytestring.

unpack

```
{Pickle.unpack +PickleV ?Value}
```

Unpacks a virtual string PickleV that has created by pickling (e.g., by `Pickle.pack`). `Pickle.unpack` may crash the Oz Engine if given a corrupt pickle.

Emulator Properties: Property

The Property module provides operations to query and possibly update Mozart system-related parameters that control various aspects of the Mozart engine and system modules.

The most important properties can be controlled graphically by means of the Mozart Panel, which is described in *"Oz Panel"*.

The properties are accessible to the programmer by operations resembling the operations on dictionaries: Property.put sets a property, whereas Property.get and Property.condGet access properties. The operations are described here (page 144) in more detail.

22.1 Engine Properties

The properties that control the Mozart engine are identified by atoms. For example, the current number of runnable threads is identified by the atom 'threads.runnable'. That is,

```
{Property.get 'threads.runnable'}
```

returns the number of currently runnable threads as an integer.

For convenience, most properties are organized into groups. A group is accessed by an atom giving the group's name ('threads', for example), and it returns a record containing the properties of that group. For example,

```
{Property.get 'threads'}
```

returns a record that has several features one of which is 'runnable'.

Some properties are read-only. They provide access to statistical information (as the property 'threads.runnable' in our previous example), but cannot be used to update that information. Other properties are mutable: changing their values customizes the engine's behaviour. For example, the property 'threads.medium' gives the ratio between the number of time slices available for threads of priorities medium and low. This can be changed to 2:1 by:

```
{Property.put 'threads.medium' 2}
```

Property.put supports groups as well. For example, to customize time slices for threads of all priorities, we can do:

```
{Property.put 'threads' foo('medium': 2
                             'high':   2)}
```

The record's label is not significant.

All properties are listed in the following sections, which are sorted by group.

Application Support: application

Field	Mutable	Type	Explanation
args	no	list of atoms	The arguments passed to an application.
url	no	Atom	The url of the root functor of an application.

Distribution: dp

Field	Mutable	Type	Explanation
`version`	no	string	The distribution version. Only ozengines with same version can communicate.
`clockTick`	no	integer	The distribution subsystem is activated at least ev `clockTick` milliseconds.
`bufferSize`	yes	integer	The size of the output and input buffers in bytes. throughput-bound applications it should be la enough to keep the network utilized between in cations of the distribution subsystem.
`probeInterval`	yes	integer	How often probes are run in milliseconds.
`probeTimeout`	yes	integer	Time before a non responded ping defines temp milliseconds.
`openTimeout`	yes	integer	Maximum time to wait for a response when est lishing a connection.
`closeTimeout`	yes	integer	Maximum time to wait for a connection to close.
`wfRemoteTimeout`	yes	integer	Maximum time to wait for a remote peer to reo a connection after being remotely closed due to l of resources.
`firewallReopenTimeout`	yes	integer	Time to wait before reopening a connection after ing remotely closed due to lack of resources wh behind a firewall.
`tcpHardLimit`	yes	integer	Maximum number of simultaneous tcp connectio
`tcpWeakLimit`	yes	integer	Number of simultaneous tcp connections when ble. The difference between tcpHardLimit and t WeakLimit is used for accepts. If they are set equ no further accepts will be made until some conn tion has been closed.
`retryTimeFloor`	yes	integer	Least time to wait before retrying a lost connecti
`retryTimeCeiling`	yes	integer	Longest time to wait before retrying a lost conn tion.
`retryTimeFactor`	yes	integer	Factor determining actual to wait before retryin lost connection.
`flowBufferSize`	yes	integer	Experimental property for flowcontrol.
`flowBufferTime`	yes	integer	Experimental property for flowcontrol.

Some of these properties affect running connections and some affect only new ones. Running: tcp*Limit. New: probe*, retryTime*.

Logging of distributed events: `dpLog`

Field	Mutable	Type	Explanation
`connectLog`	yes	`Bool`	Wheter messages and events having to do with keeping the state of the connections used by the distribution layer should be writen to the log.
`messageLog`	yes	`Bool`	Wheter messages having to do with keeping the state of the entity protocols of distributed mozart should be writen to the log.

The events are writen to standard out by default. They can be redirected to file by `DPStatistics.createLogFile`.

Printing Errors: `errors`

Field	Mutable	Type	Explanation
`debug`	yes	`Bool`	Whether error exceptions contain debug information.
`'thread'`	yes	`Int`	Number of tasks on the thread to be printed.
`width`	yes	`Int`	Maximal width used for printing values in error messages.
`depth`	yes	`Int`	Maximal depth used for printing values in error messages.
`toplevel`	yes	`Procedure`	Nullary procedure invoked after a message has been printed out about an uncaught exception raised on top level.
`subordinate`	yes	`Procedure`	Nullary procedure invoked after a message has been printed out about an uncaught exception raised within a computation space.

Finite Domains: `fd`

Field	Mutable	Type	Explanation
`variables`	no	`Int`	Number of finite domain variables created.
`propagators`	no	`Int`	Number of finite domain propagators created.
`invoked`	no	`Int`	Number of finite domain propagators invoked.
`threshold`	yes	`Int`	Integer when internal domain representation switches from bit sets to interval lists.

Garbage Collection: `gc`

Field	Mutable	Type	Explanation
`size`	no	`Int`	Current heap size in bytes
`threshold`	no	`Int`	Heap size in bytes when next automatic garbage collection takes place. Gets recomputed after every garbage collection.
`active`	no	`Int`	Heap size in bytes after last garbage collection.
`min`	yes	`Int`	Minimal heap size in bytes.
`free`	yes	`1...100`	Gives the percentage of free heap memory after garbage collection. For example, a value of 75 means that `threshold` is set to approximately: `active`*100/(100-75)=`active`*4.
`tolerance`	yes	`1...100`	Gives the percentage by which the emulator is allowed for purposes of better memory allocation to increase `threshold`.
`on`	yes	`Bool`	Whether garbage collection is invoked automatically.
`codeCycles`	yes	`Int`	After how many garbage collections also code garbage collection is performed (zero means no code garbage collection).

Implementation Limits: `limits`

Field	Mutable	Type	Explanation
`int.min`	no	`Int`	The smallest integer that can be represented efficiently (that is, by a single word in memory) by the engine.
`int.max`	no	`Int`	The largest integer that can be represented efficiently (that is, by a single word in memory) by the engine.
`bytecode.xregisters`	no	`Int`	The number of X registers this engine is able to handle in the bytecode.

Marshaler: `marshaler`

Field	Mutable	Type	Explanation
`version`	no	a hash-tuple of two integers	The version of the marshaler.

Memory Usage: memory

Field	Mutable	Type	Explanation
atoms	no	Int	Memory used in bytes for atoms.
names	no	Int	Memory used in bytes for names.
freelist	no	Int	Memory allocated but held in free lists for later use in bytes.
code	no	Int	Memory used in bytes for Mozart bytecode.
heap	no	Int	Total memory used in Kilo bytes (i.e., 1024 bytes) since start of the Mozart engine. Is increased after each garbage collection by the heap threshold.

Printing Messages: messages

Field	Mutable	Type	Explanation
gc	yes	Bool	Whether messages on garbage collection are printed.
idle	yes	Bool	Whether messages are printed when the Mozart Engine gets idle.

Platform Information: `platform`

Field	Mutable	Type	Explanation
`name`	no	`Atom`	The name of the platform as atom of the form `OS-ARCH` where the following combinations are currently supported:
`os`	no	`Atom`	The operating system part of the platform name.
`arch`	no	`Atom`	The architecture part of the platform name.

Name	OS	ARCH
`’aix3-rs6000’`	`’aix3’`	`’rs6000’`
`’freebsd-i486’`	`’freebsd’`	`’i486’`
`’freebsdelf-i486’`	`’freebsdelf’`	`’i486’`
`’irix5-mips’`	`’irix’`	`’mips’`
`’hpux-700’`	`’hpux’`	`’700’`
`’linux-arm’`	`’linux’`	`’arm’`
`’linux-i486’`	`’linux’`	`’i486’`
`’linux-m68k’`	`’linux’`	`’m68k’`
`’linux-mips’`	`’linux’`	`’mips’`
`’linux-parisc’`	`’linux’`	`’parisc’`
`’linux-ppc’`	`’linux’`	`’ppc’`
`’linux-s390’`	`’linux’`	`’s390’`
`’linux-sparc’`	`’linux’`	`’sparc’`
`’netbsd-i486’`	`’netbsd’`	`’i486’`
`’netbsd-m68k’`	`’netbsd’`	`’m68k’`
`’netbsd-sparc’`	`’netbsd’`	`’sparc’`
`’openbsd-i486’`	`’openbsd’`	`’i486’`
`’openbsd-sparc`	`’openbsd’`	`’sparc`
`’osf1-alpha’`	`’osf1’`	`’alpha’`
`’powermac-darwin’`	`’powermac’`	`’darwin’`
`’solaris-i486’`	`’solaris’`	`’i486’`
`’solaris-sparc’`	`’solaris’`	`’sparc’`
`’sunos-sparc’`	`’sunos’`	`’sparc’`
`’ultrix-mips’`	`’ultrix’`	`’mips’`
`’win32-i486’`	`’win32’`	`’i486’`

Printing Values: `print`

Field	Mutable	Type	Explanation
`verbose`	yes	`Bool`	Whether printing values includes verbose information on variables. Used for all printing routines.
`width`	yes	`Int`	Maximal width used for `System.show` and `System.print` (see Chapter 26).
`depth`	yes	`Int`	Maximal depth used for `System.show` and `System.print` (see Chapter 26).
`scientificFloats`	yes	`Bool`	Forces the scientific (with an exponent) representation of floats. By default floats are printed in the decimal notation unless the exponent of the scientific representation is smaller than -4 or larger or equal to the precision (the ’g’ conversion by the ’fprintf’ function in POSIX.1).
`floatPrecision`	yes	`Int`	Number of digits in the representation of floats, 5 by default.

Thread Priorities: `priorities`

Field	Mutable	Type	Explanation
`high`	yes	`1...100`	Relation between time slices available for threads of priorities `medium` and `low`.
`medium`	yes	`1...100`	Relation between time slices available for threads of priorities `high` and `medium`.

Computation Spaces: `spaces`

Field	Mutable	Type	Explanation
`created`	no	`Int`	Number of computation spaces created by `Space.new`.
`cloned`	no	`Int`	Number of computation spaces cloned by `Space.clone`.
`committed`	no	`Int`	Number of computation spaces committed by `Space.commit`.
`failed`	no	`Int`	Number of failed computation spaces.
`succeeded`	no	`Int`	Number of succeeded computation spaces.

Threads: `threads`

Field	Mutable	Type	Explanation
`created`	no	`Int`	Total number of threads created.
`runnable`	no	`Int`	Number of currently runnable threads.
`min`	no	`Int`	Minimal size of a thread stack in number of tasks.

Time Usage: `time`

Field	Mutable	Type	Explanation
`user`	no	`Int`	Operating system user time of the Oz Emulator process in milliseconds.
`system`	no	`Int`	Operating system system time of the Oz Emulator process in milliseconds.
`total`	no	`Int`	Elapsed real time in milli seconds from an arbitrary point in the past (for example, system start-up time). Can be used to determine the wall time elapsed between two successive applications of `{Property.get 'time.total'}`.
`run`	no	`Int`	Run time in milliseconds.
`idle`	no	`Int`	Idle time in milliseconds.
`copy`	no	`Int`	Time spent on copying (that is, on cloning of spaces) in milliseconds.
`propagate`	no	`Int`	Time spent on executing propagators in milliseconds.
`gc`	no	`Int`	Time spent on garbage collection in milliseconds.
`detailed`	yes	`Bool`	Only if **`true`**, the fields `time.copy`, `time.gc`, `time.propagate` and `time.run` are updated accordingly.

22.2 Environment Properties

Other miscellaneous properties characterize the environment in which Mozart is installed and runs.

Property	Type	Description
`oz.dotoz`	VirtualString	user directory for personal customizations and packages. Initialized from `OZ_DOTOZ` or `OZDOTOZ`, or the default `~/.oz/1.3.0`
`oz.home`	VirtualString	top directory of the mozart installation. Initialized from `OZ_HOME` or `OZHOME` or property `oz.configure.home`
`oz.search.load`	VirtualString	methods for resolving functors: initialized from `OZ_SEARCH_LOAD`, `OZ_LOAD` or `OZLOAD` (or a default)
`oz.search.path`	VirtualString	where to search for includes: initialized from `OZ_SEARCH_PATH`, `OZ_PATH` or `OZPATH` (or a default)
`oz.trace.load`	Bool	controls whether resolvers output tracing information
`path.separator`	Char	the character used as a search path separator for the current platform (`:` for Unix, and `;` for Windows)
`path.escape`	Char	the character used for escaping another character (usually backslash)
`user.home`	VirtualString	user home directory where applicable. Initialized from `HOME` (or otherwise from `HOMEDRIVE` and `HOMEPATH` on Windows)

22.3 The Programming Interface

`get`

```
{Property.get +LI X}
```

Returns the property stored under the key `LI` (a literal or an integer). Raises an exception, if no property with key `LI` exists.

`condGet`

```
{Property.condGet +LI X Y}
```

Returns the property stored under the key `LI` (a literal or an integer). If no property with key `LI` exists, `X` is returned.

`put`

```
{Property.put +LI X}
```

Stores the property `X` under key `LI` (a literal or an integer). Raises an exception, if the property is read-only.

23

Error Formatting: `Error`

The `Error` module is concerned with various tasks related to error reporting. This encompasses the following:

- Reporting errors as represented by data-structures called error messages.
- Constructing error messages from run-time error conditions in the form of exceptions.
- Registering error formatters in the error registry.

At boot time, the system installs a default exception handler processing all uncaught exceptions. This involves printing out the exception with the mechanisms mentioned above and executing a handler as given by the properties `'errors.toplevel'` and `'errors.subordinate'`, which see.

23.1 Data Structures

The central data structure used in this module is the error message. The general format is as follows:

```
⟨message⟩  ::=  ⟨message label⟩(
                    [kind: ⟨extended virtual string⟩]   % origin subsystem or compon
                    [msg: ⟨extended virtual string⟩]    % main message
                    [items: [⟨line⟩]]                   % additional information
                    ...)                                % internal fields
```

All fields of the record are optional and specify information as indicated by the comments (wherever applicable). It is recommended that both `kind` and `msg` start with a lower-case letter and do not end in a period.

The label of the record is currently ignored by the procedures from the `Error` module, but other system modules expect it to be either `error` or `warn`, depending on the severity of the condition.

```
⟨message label⟩  ::=  error | warn
```

The `items` describe a sequence of lines meant to give additional hints about the error, but one should make sure that the error message is comprehensible without this information. All keys should start with a capital letter.

```
⟨line⟩   ::=  hint([l: ⟨extended virtual string⟩]
                   [m: ⟨extended virtual string⟩])   % key/value pair
            |  ⟨coordinates⟩                          % source code error relates to
            |  line(⟨extended virtual string⟩)        % full line of text
            |  unit                                   % empty line (separator)
```

```
⟨coordinates⟩   ::=  pos(⟨atom⟩    % file name; '' if not known
                         ⟨int⟩     % line number; required
                         ⟨int⟩)    % column number; ~1 if not known
```

An ⟨extended virtual string⟩ is a virtual string that may contain, for convenience, embedded records with special interpretation.

```
⟨extended virtual string⟩   ::=  ⟨atom⟩ | ⟨int⟩ | ⟨float⟩ | ⟨string⟩
                              |  '#'(⟨extended virtual string⟩ ... ⟨extended virtual stri
                              |  oz(⟨value⟩)
                              |  pn(⟨atom⟩)
                              |  ⟨coordinates⟩
                              |  apply(⟨procedure or print name⟩ [⟨value⟩])
                              |  list([⟨value⟩] ⟨extended virtual string⟩)
```

```
⟨procedure or print name⟩   ::=  ⟨procedure⟩ | ⟨atom⟩
```

The embedded records are translated to virtual strings as follows:

1. `oz(X)` transforms `X` using `Value.toVirtualString`, using the print depth and width given by the system properties `'errors.depth'` and `'errors.width'`, respectively.

2. `pn(+A)` considers `A` to be a variable print name, i.e., escapes non-printable characters according to variable concrete syntax if `A` is enclosed in backquotes.

3. `pos(+A +I +I)` prints out source coordinates, e.g., `in file A, line I, column I` with the unspecified parts omitted.

4. `apply(+X +Ys)` represents an Oz application of `X` to `Ys`. Output uses the usual brace notation.

5. `list(+Xs +ExtendedVirtualString)` outputs the values in `Xs` as if each was wrapped inside `oz(...)`, inserting `ExtendedVirtualString` between every pair of elements.

23.2 The Error Registry

The error registry has the purpose of storing so-called error formatters under specific keys of type ‘literal’. An error formatter *P* is a procedure with the signature

```
{P +ExceptionR ?MessageR}
```

P must be capable of processing any exception having as label any of the keys under which *P* has been entered into the error registry and must return a ⟨message⟩ as defined above describing the condition. In the case of error or system exceptions, only the record found in the dispatch field of the exception is handed to the formatter.

Error formatters may be invoked by the default exception handler installed at boot time. Possibly dealing with serious conditions, formatters are required to be robust. In particular, the handler flags the thread executing the formatter to be non-blocking, i.e., if it ever blocks on a logic variable, an exception is raised in this thread. This increases chances of any message being output at all. Note that it is allowed to block on futures though and that this flag is not inherited by any created child thread.

23.3 Example Error Formatter

The following piece of code illustrates how an error formatter might be registered and how it could behave. Assume a system component called `compiler`, which is given ‘queries’ to process. If any query is ill-typed, an exception is raised, containing the query, the number of the ill-typed argument, and the expected argument type. Furthermore, an `internal` exception is raised when an internal programming assertion is violated. For robustness, an **`else`** case is included to handle any other exceptions. The formatter simply prints out the exception record, since this might help more than no output at all.

```
{Error.registerFormatter compiler
 fun {$ E}
    T = 'compiler engine error'
    BugReport = 'please send a bug report to mozart-bugs@ps.uni-sb.de'
 in
    case E of compiler(internal X) then
       error(kind: T
             msg: 'Internal compiler error'
             items: [hint(l: 'Additional information' m: oz(X))
                     line(BugReport)])
    elseof compiler(invalidQuery M I A) then
       error(kind: T
             msg: 'Ill-typed query argument'
             items: [hint(l: 'Query' m: oz(M))
                     hint(l: 'At argument' m: I)
                     hint(l: 'Expected type' m: A)])
    ...
    else
       error(kind: T
                     items: [line(oz(E))])
         end
      end}
```

23.4 The Module

1. `exceptionToMessage` {Error.exceptionToMessage Exception Message}constructs a ⟨message⟩ from an exception, using the formatters defined in the error registry or a generic formatter if none is defined for the exception. The message returned by the formatter is enriched with additional fields copied from the exception.

2. `messageToVirtualString` {Error.messageToVirtualString Message V}converts a ⟨message⟩ to a virtual string using the standard layout. This can span several lines and includes the final newline.

3. `extendedVSToVS` {Error.extendedVSToVS ExtendedVirtualString V}converts an ⟨extended virtual string⟩ to a ⟨virtual string⟩.

4. `printException` {Error.printException Exception}converts an exception to a message and this to a virtual string, printing the result on standard error (using `System.printError`).

5. `registerFormatter` {Error.registerFormatter L P}enters a formatter for exceptions with label `L` into the error registry, quietly replacing a possibly existing formatter for `L`.

System Error Formatters: `ErrorFormatters`

This module provides a set of formatters for system componeents. These formatters are registered by the `Error` module. The system components handled by this module are the following:

`kernel`

is an error formatter for exceptions coming from Kernel Oz, such as type errors.

`object`

is an error formatter for exceptions coming from the object system, e.g., inheritance errors.

`failure`

is an error formatter for exceptions pertaining to failure (i.e., inconsistent ‘tell’ operations on the store).

`recordC`

is an error formatter for exceptions coming from the feature constraint system.

`system`

is an error formatter for exceptions coming from libraries used for system programming.

`ap`

is an error formatter for exceptions coming from libraries used for application programming.

`dp`

is an error formatter for exceptions coming from libraries used for distributed programming.

`os`

is an error formatter for exceptions coming from the operating system interface.

`foreign`

is an error formatter for exceptions coming from foreign function interface.

`url`

is an error formatter for exceptions coming from the URL resolving library.

`module`

is an error formatter for exceptions coming from the module manager.

25

Memory Management: `Finalize`

The finalization facility gives the programmer the ability to process an Oz value when it is discovered (during garbage collection) that it has otherwise become unreachable. This is often used to release native resources held or encapsulated by said value.

Native functors and extension classes make it very easy to extend Oz with new interfaces to arbitrary resources. Typically, the access to a resource will be encapsulated into a data-structure that is an instance of an extension class and we would like the resource to be automatically released when the data-structure is no longer being referenced and can be garbage collected. For example:

- we may encapsulate a handle to access a database: when the handle is garbage collected, we would like the connection to the database to be automatically closed.
- we may encapsulate a pointer to malloc'ed memory, e.g. for a large bitmap to be manipulated by native DLLs: when this is no longer referenced, we would like the memory to be automatically freed.

This is the purpose of *finalization*: to execute a cleanup action when a computational object is no longer referenced. The Oz finalization facility was inspired by the article Guardians in a Generation-Based Garbage Collector[1] (R. Kent Dybvig, Carl Bruggeman, David Eby, June 1993). It is built on top of weak dictionaries (Section *Weak Dictionaries*, *(The Oz Base Environment)*): each weak dictionary is associated with a *finalization stream* on which pairs `Key#Value` appear when `Value` becomes unreachable except through one or more weak dictionaries.

Note that since instances of the same stateless value are indistinguishable, we cannot tell the difference between one instance of the integer 7 and another one. Thus, the question of whether such an instance is unreachable or whether it is an identical copy of it which is unreachable can usually not be answered. Also, since these values are indistinguishable, the engine is in principle licensed to either duplicate or merge them at its discretion (duplication actually occurs when such values are distributed to other sites). For this reason, finalization only makes sense for datatypes with token equality, i.e. where the identity of an instance is unique. Stateless values with structural equality are treated by the finalization mechanism as if they were unreachable (i.e. as if the weak dictionary contained a copy) and will be subjected to finalization at the very next garbage collection. One exception are atoms: since they remain in the global atom

[1] `ftp://ftp.cs.indiana.edu/pub/scheme-repository/doc/pubs/guardians.ps.gz`

table, they are currently considered to always be reachable. This might change if we implement garbage collection of the atom table.

Finalize.guardian

```
{Finalize.guardian +Finalizer ?Register}
```

This takes as input a 1-ary procedure `Finalizer` and returns a 1-ary procedure `Register` representing a new guardian. A value `X` can be registered in the guardian using:

```
{Register X}
```

Thereafter, when `X` becomes unreachable except through a weak dictionary (e.g. a guardian), at the next garbage collection `X` is removed from the guardian and the following is eventually executed:

```
{Finalizer X}
```

We say eventually because the finalization thread is subject to the same fair scheduling as any other thread. Note that the value `X` has still not been garbage collected; only at the next garbage collection after the call to `Finalizer` has returned, and assuming `Value` is no longer referenced at all, not even by the guardian, will it really be garbage collected.

The `Register` procedure cannot be used in a (subordinated) space other than where `Finalize.guardian` has been called. Keep in mind that it probably doesn't make sense to create a guardian in a non-toplevel space because a space can become failed. When this happens, everything in the space is discarded, including the guardians, which means the values registered in them will never be finalized.

Finalize.register

```
{Finalize.register +Value +Handler}
```

This is a slightly different interface that allows to register simultaneously a `Value` and a corresponding finalizer procedure `Handler`. After `Value` becomes otherwise unreachable, `{Handler Value}` is eventually executed.

Finalize.everyGC

```
{Finalize.everyGC +P/0}
```

This simply registers a 0-ary procedure to be invoked after each garbage collection. Note that you cannot rely on how soon after the garbage collection this procedure will really be invoked: it is in principle possible that the call may only be scheduled several garbage collections later if the system has an incredibly large number of live threads and generates tons of garbage. It is instructive to look at the definition of `EveryGC`:

```
proc {EveryGC P}
   proc {DO _} {P} {Finalize.register DO DO} end
in {Finalize.register DO DO} end
```

in other words, we create a procedure `DO` and register it using itself as its own handler. When invoked, it calls `P` and registers itself again.

26

Miscelleanous System Support: `System`

The `System` module contains procedures providing functionality related to the Mozart Engine.

26.1 System Control

`gcDo`

```
{System.gcDo}
```

Invokes garbage collection.

26.2 Printing

The procedures to print values and virtual strings choose the output device for the printed text as follows:

1. If Mozart is running standalone, the standard output or standard error device is chosen (depending on the procedure).
2. Otherwise the Oz Programming Interface is chosen as output device.

The procedures explained here differ from those provided by `Open` and `OS` in that they can be used to print information in subordinated computation spaces to support debugging.

`print`

```
{System.print X}
```

The current information on X is printed without a following newline.

The output is limited in depth and width by system parameters that can be configured either by the Oz Panel (see *"Oz Panel"*) or by `Property.put` (see Chapter 22).

A width of *n* means that for lists at most *n* elements and for records at most *n* fields are printed, the unprinted elements and fields are abbreviated by „,. A depth of *n* means that trees are printed to a depth limit of *n* only, deeper subtrees are abbreviated by „,.

The printed text appears on standard output.

`show`

`{System.show X}`

The current information on X is printed with a following newline.

The output is limited in depth and width as with `System.print`.

The printed text appears on standard output.

`printError`

`{System.printError V}`

Prints the virtual string V without a newline.

The printed text appears on standard error.

`showError`

`{System.showError V}`

Prints the virtual string V followed by a newline.

The printed text appears on standard error.

`printInfo`

`{System.printInfo V}`

Prints the virtual string V without a newline.

The printed text appears on standard output.

`showInfo`

`{System.showInfo V}`

Prints the virtual string V followed by a newline.

The printed text appears on standard output.

26.3 Miscellaneous

`eq`

`{System.eq X Y ?B}`

Tests whether X and Y refer to the same value node in the store.

`nbSusps`

`{System.nbSusps X ?I}`

Returns the number of suspensions on X, that is the number of threads and propagators that suspend on X.

`onToplevel`

`{System.onToplevel ?B}`

Returns **`true`** iff the current thread is executing in the top level space and not within a deep space.

Part VI

Window Programming

27

The Module Tk

This chapter contains reference information for the `Tk` module.

27.1 Tickles

Tickles are Oz values used to communicate with the graphics engine. The graphics engine receives and executes tickles. The graphics engine is implemented in Tcl/Tk (see [6]). In order to execute tickles the graphics engine first translates tickles into strings. This section defines tickles, defines how tickles are translated into strings, and presents the Oz procedures to send tickles.

27.1.1 Syntax

The set of tickles contains virtual strings, boolean values, and so-called tickle-objects. A tickle-object is an object which is created from a class the Tk module provides for (all classes but `Tk.listener`). Roughly spoken, the set of tickles is closed under record construction, where only records are allowed which do not contain names as features or as label. Proper records with the labels `v`, `b`, `#`, and `|` are special cases. Examples for tickles can be found in Section *The Graphics Engine, Tickles, and Widget Messages, (Window Programming in Mozart)*.

The exact definition of a tickle is given by the procedure `IsTcl` which is shown in Figure 27.1. The procedure `IsTcl` returns **`true`**, if and only if `X` is a tickle. Otherwise **`false`** is returned. The procedure `IsTclObject` tests whether an object is a tickle-object. Note that records which have the labels `#` and `|` are treated as virtual strings. Note that `IsTcl` and the following procedures serve as specification, the graphics engine itself employs well optimized routines instead.

27.1.2 Translation to Virtual Strings

The translation of a tickle into a virtual string that then by the graphics agent is interpreted as a tcl command is shown in Figure 27.2. The used help routines are shown in Figure 27.3.

Figure 27.1 Procedure `IsTcl` tests whether a value is tickle.

```
fun {IsTcl X}
   {IsBool X} orelse {IsUnit X} orelse
   {IsVirtualString X} orelse
   {IsTclObject X} orelse
   {IsRecord X} andthen
   {Not {Some {Arity X} IsName}} andthen
   {Not {IsName {Label X}}} andthen
   case {Label X}
   of  v  then {Arity X}==[1] andthen {IsVirtualString X.1}
   []  b  then {Arity X}==[1] andthen {All X.1 IsTcl}
   []  c  then {Arity X}==[1 2 3] andthen
               {All X fun {$ I} I>=0 andthen I<=255 end}
   [] '#' then false
   [] '|' then false
   else {Record.all X IsTcl}
   end
end
```

27.1.3 Sending Tickles

Tickles can be send to the graphics engine with the following procedures. The graphics engine processes tickles in batches: it reads a batch of tickles and executes it. If no further batch can be read currently, it updates the graphics. After having updated the graphics, it checks whether user events are to be processed.

The Oz procedures to send tickles are asynchronous and preserve order: all tickles are processed in the same order they are send in. However, after the procedure has been executed, the graphics engine might not yet have executed the tickle.

send

`{Tk.send +Tcl}`

Sends +`Tcl` to the graphics engine.

batch

`{Tk.batch +TclS}`

Sends a list of list of tickles +`TclS` to the graphics engine. It is guaranteed that the graphics engine processes all tickles in `TclS` in a single batch.

27.1.4 Sending Tickles and Returning Values

In the same way as sending tickles to graphics engine, the engine can asynchronously send back return values which are strings. The following procedures send tickles and return the values returned by executing the tickles by the graphics engine.

returnString

`{Tk.returnString +Tcl ?S}`

Figure 27.2 Procedure `TclToV` translates a tickle into a virtual string.

```
local
   fun {FieldToV AI Tcl}
      if {IsInt AI} then '' else '-'#{Quote AI}#' ' end # {TclToV Tcl}
   end
   fun {RecordToV R AIs}
      {FoldR AIs fun {$ AI V}
                    {FieldToV AI R.AI} # ' ' # V
                 end ''}
   end
in
   fun {TclToV Tcl}
      if {IsBool Tcl} then case Tcl then 0 else 1 end
      elseif {IsUnit Tcl} then ''
      elseif {IsVirtualString Tcl} then {Quote Tcl}
      elseif {IsTclObject Tcl} then {TclObjectToV Tcl}
      else
        case {Label Tcl}
        of o then {RecordToV Tcl {Arity Tcl}}
        [] p then AI|AIs={Arity Tcl} in
           '{'#{FieldToV AI Tcl.AI}#'.'#{RecordToV Tcl AIs}#'}'
        [] b then {FoldR Tcl.1 fun {$ Tcl V}
                                  {TclToV Tcl}#' '#V
                               end ''}
        [] c then '#'#{Hex Tcl.1}#{Hex Tcl.2}#{Hex Tcl.3}
        [] v then Tcl.1
        [] s then '"'#{RecordToV Tcl {Arity Tcl}}#'"'
        [] l then '['#{RecordToV Tcl {Arity Tcl}}#']'
        [] q then '{'#{RecordToV Tcl {Arity Tcl}}#'}'
        else {Quote {Label Tcl}}#' '#{RecordToV Tcl {Arity Tcl}}
        end
      end
   end
end
```

Figure 27.3 Help routines to translate a tickle into a virtual string.

160a ⟨**Definition of Octal** 160a⟩≡

```
fun {Octal I}
   [&\\ (I div 64 + &0) ((I mod 64) div 8 + &0) (I mod 8 + &0)]
end
```

160b ⟨**Definition of Quote** 160b⟩≡

```
fun {Quote V}
   case {VirtualString.toString V} of nil then "\"\""
   [] S then
      {FoldR S fun {$ I Ir}
                  if {Member I "{}[]\\$\";"} then &\\|I|Ir
                  elseif I<33 orelse I>127 then {Append {Octal I} Ir}
                  else I|Ir
                  end
               end nil}
   end
end
```

160c ⟨**Definition of Hex** 160c⟩≡

```
local
   fun {HexDigit I}
      I + if I>9 then &a-10 else &0 end
   end
in
   fun {Hex I}
      [{HexDigit I div 16} {HexDigit I mod 16}]
   end
end
```

Returns the result of sending and executing +`Tcl` as string.

`return`

`{Tk.return +Tcl ?S}`

Shortcut for `Tk.returnString`.

`returnAtom`

`{Tk.returnAtom +Tcl ?A}`

Returns the result of sending and executing +`Tcl` as string.

`returnInt`

`{Tk.returnInt +Tcl ?IB}`

Returns the result of sending and executing +`Tcl` as integer. If the result does not describe a number `false` is returned.

`returnFloat`

`{Tk.returnFloat +Tcl ?FB}`

Returns the result of sending and executing +`Tcl` as float. If the result does not describe a number `false` is returned.

`returnListString`

`{Tk.returnListString +Tcl ?Ss}`

Returns the result of sending and executing `+Tcl` as list of strings.

`returnList`

`{Tk.returnList +Tcl ?Ss}`

Shortcut for `Tk.returnListString`.

`returnListAtom`

`{Tk.returnListAtom +Tcl ?ABs}`

Returns the result of sending and executing `+Tcl` as list of atoms. If elements of the list do not form valid atoms, the list contains the element **`false`** instead.

`returnListInt`

`{Tk.returnListInt +Tcl ?IBs}`

Returns the result of sending and executing `+Tcl` as list of integers. If elements of the list do not form valid numbers, the list contains the element **`false`** instead.

`returnListFloat`

`{Tk.returnListFloat +Tcl ?FBs}`

Returns the result of sending and executing `+Tcl` as list of integers. If elements of the list do not form valid numbers, the list contains the element **`false`** instead.

27.2 Tickle Objects

Rather than programming with tickles directly, all graphical entities are provided as classes. To these classes we refer to as tickle classes and to their instances as tickle objects. Applying a tickle object to a message translates it in a straightforward way into a tickle.

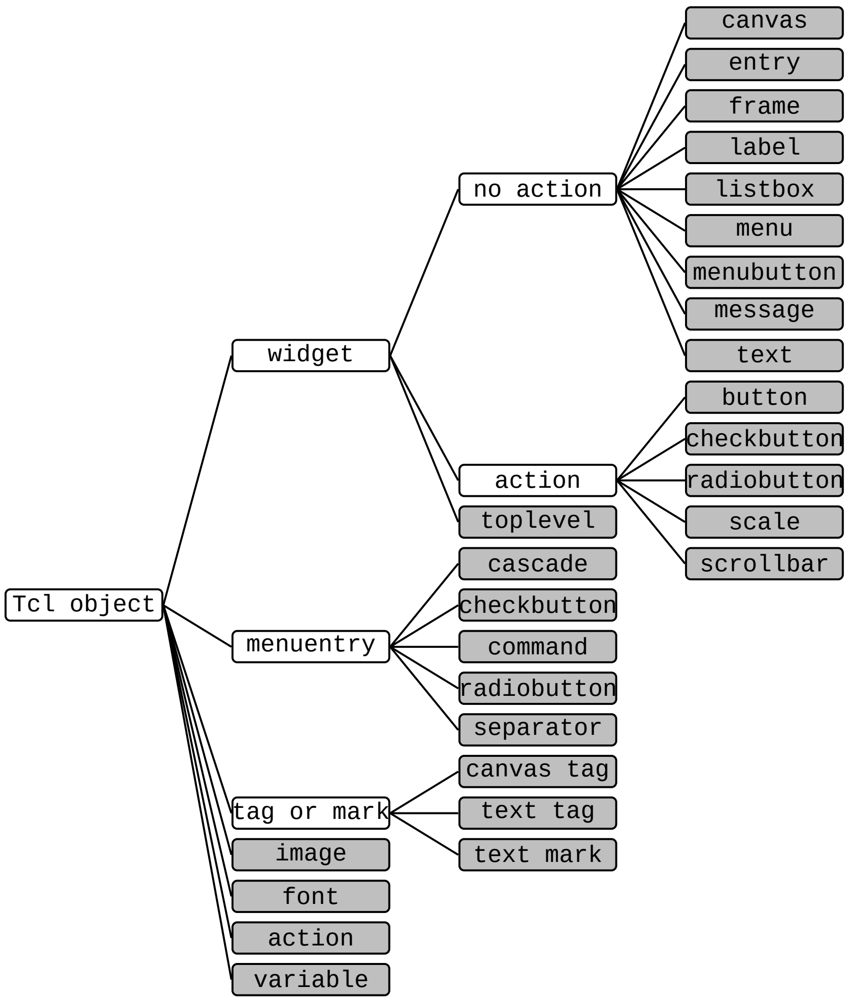
Tickle objects themselves enjoy an translation into strings, each tickle object carries a unique identifier that is its translation. Creation of a tickle object creates a new unique identifier, where the exact identifier depends on the tickle class the object is created from. The translation of a tickle object application of course uses the tickle object's identifier.

The available tickle classes are shown in Figure 27.4, where the classes that are shadowed gray are classes provided by the `Tk` module, the others are just conceptual. The rest of the document will be concerned with giving a short overview on the tickle object interfaces.

Messages to tickle objects are translated in a straightforward way to tickles, for examples see Section *The Graphics Engine, Tickles, and Widget Messages, (Window Programming in Mozart)*.

In the following we will present values other than tickles that are used in messages to tickle objects.

Figure 27.4 Hierarchy of tickle objects.



27.2.1 Action Values

An action value is either

A procedure `P`**.** The action is invoked by creating a new thread that applies `P` to action or event arguments.

A listener object method pair `ListenerO#M`

The action is invoked by letting the `ListenerO` serve the message `M1` (see Section 27.11). The message `M1` is obtained by appending action or event arguments to `M`.

An object method pair `O#M`

Here of course, `O` must not be an instance of a listener. The action is invoked by creating a new thread that executes the object application `{O M1}`. The message `M1` is obtained by appending action or event arguments to `M`.

27.2.2 Action Argument Values

When an action is executed, it possibly has some arguments attached to it. Action argument values describe the number and types of arguments that are used with the invoked action value. For an example see Section *Scales*, *(Window Programming in Mozart)*.

Legal action argument values are lists whose elements are drawn from the following set:

`string`	`atom`	`int`	`float`
`list(string)`	`list(atom)`	`list(int)`	`list(float)`

The length of the list defines the number of action arguments and the elements define in the obvious way the type of the arguments.

27.2.3 Event Argument Values

When an event is executed, it possibly has some arguments attached to it. Event argument values describe the number, types, and substitutions of arguments that are used with the invoked action value. For an example see Section *Canvas Tags*, *(Window Programming in Mozart)*. For reference information on substitutions see `bind`[1].

Legal action argument values are lists whose elements are drawn from the following set:

`string(V)`	`atom(V)`	`int(V)`	`float(V)`
`list(string(V))`	`list(atom(V))`	`list(int(V))`	`list(float(V))`

The length of the list defines the number of action arguments and the elements define the type of the arguments, where V gives the substitution to be performed. For example, the value `[int(x)]` describes, that the action handler value takes a single argument. The single argument will be `x`-coordinate as an integer associated with the event.

27.3 No-Action Widgets

The following table lists the widgets that do not accept actions:

[1] `../tcltk/TkCmd/bind.htm`

Class	Example
Tk.canvas	Chapter *Canvas Widgets, (Window Programming in Mozart)*
Tk.entry	Section *Entries and Focus, (Window Programming in Mozart)*
Tk.frame	Section *Frames, (Window Programming in Mozart)*
Tk.label	Section *Label Widgets, (Window Programming in Mozart)*
Tk.listbox	Section *Listboxes and Scrollbars, (Window Programming in Mozart)*
Tk.menu	Section *Menus, Menuitems, and Menubuttons, (Window Programming in Mozart*
Tk.menubutton	Section *Menus, Menuitems, and Menubuttons, (Window Programming in Mozart*
Tk.message	Section *Messages, (Window Programming in Mozart)*
Tk.text	Chapter *Text Widgets, (Window Programming in Mozart)*

[a] ../tcltk/TkCmd/canvas.htm
[b] ../tcltk/TkCmd/entry.htm
[c] ../tcltk/TkCmd/frame.htm
[d] ../tcltk/TkCmd/label.htm
[e] ../tcltk/TkCmd/listbox.htm
[f] ../tcltk/TkCmd/menu.htm
[g] ../tcltk/TkCmd/menubutton.htm
[h] ../tcltk/TkCmd/message.htm
[i] ../tcltk/TkCmd/text.htm

All above mentioned classes provide the following methods:

tkInit

```
tkInit(parent:+WidgetO ...)
```

Initializes the widget object and creates the widget.

The field `WidgetO` for the special feature `parent` must be a tickle object. **`self`** becomes a slave of `WidgetO`.

tk

```
tk(...)
```

Sends a command for **`self`** to the graphics engine.

tkReturn

```
tkReturn(... ?S)
tkReturnString(... ?S)
tkReturnAtom(... ?A)
tkReturnInt(... ?IB)
tkReturnFloat(... ?FB)
tkReturnList(... ?Ss)
tkReturnListString(... ?Ss)
tkReturnListAtom(... ?As)
tkReturnListInt(... ?IBs)
tkReturnListFloat(... ?FBs)
```

Sends a command for **`self`** to the graphics engine and returns the result in the field with the highest integer feature.

tkBind

```
tkBind(event:  EventV
       action: Action  <= _
       args:   Args    <= nil
```

```
          append: AppendB <= false
          break:  BreakB  <= false)
```

Defines an event binding for `self`. Events are described in Section *Events, (Window Programming in Mozart)* and Section 27.2.3.

`tkClose`

```
tkClose()
```

Closes the widget object and destroys the widgets and all slave widgets. Further object application of `self` raises an exception.

27.4 Action Widgets

Class	Example
`Tk.button`	Section *Buttons and Actions, (Window Programming in Mozart)*
`Tk.checkbutton`	Section *Checkbuttons, Radiobuttons, and Variables, (Window Programming in M*
`Tk.radiobutton`	Section *Checkbuttons, Radiobuttons, and Variables, (Window Programming in M*
`Tk.scale`	Section *Scales, (Window Programming in Mozart)*
`Tk.scrollbar`	Section *Listboxes and Scrollbars, (Window Programming in Mozart)*

[a] `../tcltk/TkCmd/button.htm`
[b] `../tcltk/TkCmd/checkbutton.htm`
[c] `../tcltk/TkCmd/radiobutton.htm`
[d] `../tcltk/TkCmd/scale.htm`
[e] `../tcltk/TkCmd/scrollbar.htm`

`tkInit`

```
tkInit(parent: +WidgetO
       action: +Action
       args:   +Args   <= nil
       ...)
```

Initializes the widget object and creates the widget.

The field `WidgetO` for the special feature `parent` must be a tickle object. `self` becomes a slave of `WidgetO`. `Action` and `Args` are described in Section 27.2.1 and Section 27.2.2.

`tk`

```
tk(...)
```

Sends a command for `self` to the graphics engine.

`tkReturn`

```
tkReturn(... ?S)
tkReturnString(... ?S)
tkReturnAtom(... ?A)
tkReturnInt(... ?IB)
tkReturnFloat(... ?FB)
tkReturnList(... ?Ss)
tkReturnListString(... ?Ss)
tkReturnListAtom(... ?As)
tkReturnListInt(... ?IBs)
tkReturnListFloat(... ?FBs)
```

Sends a command for **self** to the graphics engine and returns the result in the field with the highest integer feature.

tkAction

```
tkAction(action: +Action <= unit
         args:   +Args)
```

Redefines or deletes an action. `Action` and `Args` are described in Section 27.2.1 and Section 27.2.2.

tkBind

```
tkBind(event:  EventV
       action: Action  <= _
       args:   Args    <= nil
       append: AppendB <= false
       break:  BreakB  <= false)
```

Defines an event binding for **self**. Events are described in Section *Events, (Window Programming in Mozart)* and Section 27.2.3.

tkClose

```
tkClose()
```

Closes the widget object and destroys the widgets and all slave widgets. Further object application of **self** raises an exception.

27.5 Toplevel Widgets

The class `Tk.toplevel` provides the following methods:

tkInit

```
tkInit(parent:   +ParentTcl
       delete:   +DeleteAction
       title:    +TitleTcl
       withdraw: +WithdrawB
       ...)
```

Initializes the widget object and creates a new toplevel widget.

For more information see Section *Toplevel Widgets and Widget Objects, (Window Programming in Mozart)* and Section *Toplevel Widgets and Window Manager Commands, (Window Programming in Mozart)*

tk

```
tk(...)
```

Sends a command for **self** to the graphics engine.

tkReturn

```
tkReturn(... ?S)
tkReturnString(... ?S)
tkReturnAtom(... ?A)
tkReturnInt(... ?IB)
tkReturnFloat(... ?FB)
```

```
tkReturnList(... ?Ss)
tkReturnListString(... ?Ss)
tkReturnListAtom(... ?As)
tkReturnListInt(... ?IBs)
tkReturnListFloat(... ?FBs)
```

Sends a command for **self** to the graphics engine and returns the result in the field with the highest integer feature.

tkBind

```
tkBind(event:  EventV
       action: Action  <= _
       args:   Args    <= nil
       append: AppendB <= false
       break:  BreakB  <= false)
```

Defines an event binding for **self**. Events are described in Section *Events, (Window Programming in Mozart)* and Section 27.2.3.

tkClose

```
tkClose()
```

Closes the widget object and destroys the widgets and all slave widgets. Further object application raises an exception.

27.6 Menu Entries

Class	Example
Tk.menuentry.cascade	Section *Menus, Menuitems, and Menubuttons, (Window Programm*
Tk.menuentry.checkbutton	Section *Menus, Menuitems, and Menubuttons, (Window Programm*
Tk.menuentry.command	Section *Menus, Menuitems, and Menubuttons, (Window Programm*
Tk.menuentry.radiobutton	Section *Menus, Menuitems, and Menubuttons, (Window Programm*
Tk.menuentry.separator	Section *Menus, Menuitems, and Menubuttons, (Window Programm*

[a]../tcltk/TkCmd/menu.htm
[b]../tcltk/TkCmd/menu.htm
[c]../tcltk/TkCmd/menu.htm
[d]../tcltk/TkCmd/menu.htm
[e]../tcltk/TkCmd/menu.htm

tkInit

```
tkInit(parent: +WidgetO
       before: +BeforeTcl
       action: +Action
       args:   +Args   <= nil
       ...)
```

Initializes and creates a menu entry.

The field `WidgetO` for the special feature `parent` must be an instance of `Tk.menu`. **self** becomes a slave of `WidgetO`. `Action` and `Args` are described in Section 27.2.1 and Section 27.2.2.

tk

```
tk(...)
```

Sends a command for `self` to the graphics engine.

`tkClose`

```
tkClose()
```

Closes the object and destroys the menu entry. Further object application raises an exception.

27.7 Variables

The class `Tk.variable` provides the following methods. Note that it does not provide a method for closing.

`tkInit`

```
tkInit(+Tcl <= unit)
```

Creates a new tickle variable with initial value `Tcl`.

`tkSet`

```
tkSet(+Tcl)
```

Sets the variable's value to +`Tcl`.

`tkReturn`

```
tkReturn(... ?S)
tkReturnString(... ?S)
tkReturnAtom(... ?A)
tkReturnInt(... ?IB)
tkReturnFloat(... ?FB)
tkReturnList(... ?Ss)
tkReturnListString(... ?Ss)
tkReturnListAtom(... ?As)
tkReturnListInt(... ?IBs)
tkReturnListFloat(... ?FBs)
```

Returns the current value of the variable.

27.8 Tags And Marks

The following table lists the widgets that do not accept actions:

Class	Example	Reference
`Tk.canvasTag`	Section *Canvas Tags, (Window Programming in Mozart)*	`canvas`[a]
`Tk.textTag`	Section *Text Tags and Marks, (Window Programming in Mozart)*	`text`[b]
`Tk.textMark`	Section *Text Tags and Marks, (Window Programming in Mozart)*	`text`[c]

[a] `../tcltk/TkCmd/canvas.htm`
[b] `../tcltk/TkCmd/text.htm`
[c] `../tcltk/TkCmd/text.htm`

The above mentioned classes provide the following methods:

tkInit

```
tkInit(parent:+WidgetO ...)
```

Initializes the widget object and creates the widget.

The field `WidgetO` for the special feature `parent` must be a tickle object. If **`self`** is an instance of `Tk.canvasTag`, `WidgetO` must be an instance of `Tk.canvas`. Otherwise, `WidgetO` must be an instance of `Tk.text`. **`self`** becomes a slave of `WidgetO`.

tk

```
tk(...)
```

Sends a command for **`self`** to the graphics engine.

tkReturn

```
tkReturn(... ?S)
tkReturnString(... ?S)
tkReturnAtom(... ?A)
tkReturnInt(... ?IB)
tkReturnFloat(... ?FB)
tkReturnList(... ?Ss)
tkReturnListString(... ?Ss)
tkReturnListAtom(... ?As)
tkReturnListInt(... ?IBs)
tkReturnListFloat(... ?FBs)
```

Sends a command for **`self`** to the graphics engine and returns the result in the field with the highest integer feature.

tkBind

```
tkBind(event:  EventV
       action: Action  <= _
       args:   Args    <= nil
       append: AppendB <= false
       break:  BreakB  <= false)
```

Defines an event binding for **`self`**. Events are described in Section *Events, (Window Programming in Mozart)*, Section *Canvas Tags, (Window Programming in Mozart)*, Section *Example: A ToyText Browser, (Window Programming in Mozart)*, and Section 27.2.3.

tkClose

```
tkClose()
```

Closes the object and performs a `delete` command on all entities that are currently referred to by **`self`**.

27.9 Images

The class `Tk.image` provides the following methods. An example can be found in Section *Images, (Window Programming in Mozart)*, for reference documentation see `image`[2].

[2] `../tcltk/TkCmd/image.htm`

tkInit

```
tkInit(type:    +TypeTcl
       url:     +UrlV <= unit
       maskurl: +MarkUrlV <= unit
       ...)
```

Creates a new image.

tk

```
tk(...)
```

Sends a command for `self` to the graphics engine.

tkReturn

```
tkReturn(... ?S)
tkReturnString(... ?S)
tkReturnAtom(... ?A)
tkReturnInt(... ?IB)
tkReturnFloat(... ?FB)
tkReturnList(... ?Ss)
tkReturnListString(... ?Ss)
tkReturnListAtom(... ?As)
tkReturnListInt(... ?IBs)
tkReturnListFloat(... ?FBs)
```

Sends a command for `self` to the graphics engine and returns the result in the field with the highest integer feature.

tkClose

```
tkClose()
```

Destroys the image and closes the image object.

27.10 Fonts

The class `Tk.font` provides the following methods. An example can be found in Section *Font Options*, *(Window Programming in Mozart)*, for reference documentation see `font`[3].

tkInit

```
tkInit(...)
```

Creates a new font.

tk

```
tk(...)
```

Sends a command for `self` to the graphics engine.

tkReturn

[3] `../tcltk/TkCmd/font.htm`

```
tkReturn(... ?S)
tkReturnString(... ?S)
tkReturnAtom(... ?A)
tkReturnInt(... ?IB)
tkReturnFloat(... ?FB)
tkReturnList(... ?Ss)
tkReturnListString(... ?Ss)
tkReturnListAtom(... ?As)
tkReturnListInt(... ?IBs)
tkReturnListFloat(... ?FBs)
```

Sends a command for `self` to the graphics engine and returns the result in the field with the highest integer feature.

tkClose

```
tkClose()
```

Destroys the font and closes the font object.

27.11 Listeners

The class `Tk.listener` is not a tcl object. Is main use is to serve as listener for events, see also Section 27.2.1. It provides the following methods. For an example see Section *More on Actions: Listeners, (Window Programming in Mozart)*.

tkInit

```
tkInit()
```

Creates a new listener, which consists of a message stream and a thread that serves each message on the stream by applying `self` to it.

tkServe

```
tkServe(+MessageR)
```

Adds the message `MessageR` to the current tail of the message stream.

tkClose

```
tkClose()
```

Decouples the message stream from the serving thread. That is, after all messages that are currently on the stream have been served, the serving thread terminates.

27.12 Strings

toAtom

```
{Tk.string.toAtom +S ?A}
```

Returns an atom corresponding to the string `S`.

toInt

```
{Tk.string.toInt +S ?IB}
```

Returns an integer corresponding to the string S. If S does not form a valid number, **false** is returned.

toFloat

```
{Tk.string.toFloat +S ?FB}
```

Returns a float corresponding to the string S. If S does not form a valid number, **false** is returned.

toListString

```
{Tk.string.toListString +S ?Ss}
```

Returns a list of strings that corresponds to the space separated substrings of S.

toListAtom

```
{Tk.string.toListAtom +S ?As}
```

Returns a list of atoms that corresponds to the space separated substrings of S.

toListInt

```
{Tk.string.toListInt +S ?IBs}
```

Returns a list of integers that corresponds to the space separated number strings of S. If one of the substrings does not form a valid number, the element will be **false** rather than an integer.

toListFloat

```
{Tk.string.toListFloat +S ?FBs}
```

Returns a list of floats that corresponds to the space separated number strings of S. If one of the substrings does not form a valid number, the element will be **false** rather than a float.

27.13 Miscellaneous

isColor

```
Tk.isColor
```

Is **true**, if the current display supports colors.

Is defined as

```
Tk.isColor={Tk.returnInt winfo(depth '.')}>1
```

addXScrollbar

```
{Tk.addXScrollbar +BarTcl +ToScrollTcl}
```

Attaches a horizontal scrollbar defined by BarTcl to a scrollable widget defined by ToScrollTcl.

addYScrollbar

```
{Tk.addYScrollbar +BarTcl +ToScrollTcl}
```

Attaches a vertical scrollbar defined by BarTcl to a scrollable widget defined by ToScrollTcl.

getId

```
{Tk.getId V}
```

Returns a virtual string V that can be used as new unique tickle identifier.

getPrefix

```
{Tk.getPrefix V}
```

Returns a virtual string V that can be used as prefix for new unique tickle identifiers. It is guaranteed that appending integers to V yields unique identifiers.

28

Graphical Tools: TkTools

28.1 Error

The `TkTools.error` class extends `TkTools.dialog`.

tkInit

```
tkInit(title:  +TitleTcl  <= 'Error'
       master: +MasterTcl <= NoArg
       aspect: +AspectI   <= 250
       text:   +TextTcl)
```

Initialise the widget object and create a new error dialog.

+AspectI specifies a non-negative integer value indicating desired aspect ratio for the text. The aspect ratio is specified as 100*width/height. 100 means the text should be as wide as it is tall, 200 means the text should be twice as wide as it is tall, 50 means the text should be twice as tall as it is wide, and so on.

28.2 Dialog

The `TkTools.dialog` class extends `Tk.frame`.

tkInit

```
tkInit(title:   +TitleTcl
       master:  +WidgetO       <= NoArg
       root:    +Root          <= master
       buttons: +ButtonsL
       pack:    +PackB         <= true
       focus:   +FocusI        <= 0
       bg:      +BackgroundA   <= NoArg
       default: +DefaultI      <= 0
       delete:  +DeleteAction  <= NoArg)
```

Initialise the widget object and create a new dialog. This widget needs at least two informations: the title and a list of buttons. Each element of the button list is of the form: `LabelTcl # nullary procedure`. Two examples are:

- `'close' # tkClose`
- `'nothing'# proc {$} skip end`

A master widget is specified by +`WidgetO`. Depending on +`Root` the widget will be placed in relation to the master like following:

`master`
The dialog will be placed in the upper left corner within the master widget.

`master#XOff#YOff`
The dialog will be placed in the upper left corner of the master widget plus the xoffset and yoffset.

`pointer`
The dialog will be placed on the current pointer position.

`X#Y`
User defined position.

If you want to use a focus then you describe by +`FocusI` the button that gets the focus. By +`DefaultI` you set the default button.

`tkPack`

```
tkPack
```

Pack the dialog.

`tkClose`

```
tkClose
```

Close the dialog.

28.3 Menubar

```
{TkTools.menubar +PWidgetO +KBWidgetO  +L +R ?Widget0}
```

The `TkTools.menubar` function creates a menubar widget. The parent is described by +`PWidgetO`, while the KeyBinder is specified by +`KBWidgetO`. Usually both the parent and the key binder are the toplevel window. If you press a key combination, that is an abbreviation for a menu entry, within the key binder area the associated action will be done.

Menu entries are specified by the lists +`L` and +`R`, one for the left and one for the right menu part. Each element of both lists is a record with the label `menubutton`. The record has four features: `text`, `underline`, `menu` and `feature`.

```
menubutton(text:      +TextTcl
           underline: +ULI
           menu:      +MenuL
           feature:   +Atom)
```

The field under the feature `menu` describes the view of the menu. It is a list of menu entries. Each entry may be: `command`, `separator`, `ckeckbutton`, `radiobutton` or cascade.

The `separator` is just an Atom. While `checkbutton` and `radiobutton` are usual Tk widgets, the `command` button is a new element and has the following form:

```
command(label:   +TextTcl
        action:  +Action
        key:     +KeyTcl
        feature: +Atom)
```

The value for the key describes the keyboard accelerator and event binding to be created. They can be used as follows:

key option value	accelerator	event binding
a	a	a
ctrl(a)	C-a	<Control-a>
alt(a)	A-a	<Alt-a>
alt(ctrl(a))	A-C-a	<Alt-Control-a>
ctrl(alt(a))	C-A-a	<Control-Alt-a>

If you like to place a submenu you have to describe a `cascade` element, which consists only of the two features `label` and `menu`.

```
cascade(label: +TextTcl
        menu:  +MenuL)
```

28.4 Popup Menu

The `TkTools.popupmenu` class extends `Tk.menubutton`.

tkInit

```
meth tkInit(parent:   +WidgetO
            entries:  +EL      <= [empty]
            selected: +SelI    <= 1
            font:     +FontO   <= unit
            action:   +ActionP <= proc {$ S} skip end ...)
```

Initialise the widget object and create a new popup menu. You always need to declare the parent of the widget. The field under the feature `entries` is a list of popup menu entries. Each entry is of the form: `LabelAtom#ColorAtom`. Although it is possible to declare an entry as `[push#black push#green]`, you should not do this.

By +SelI you declare the initial entry. If action should be performed you have to declare a unary procedure.

getCurrent

```
getCurrent(?Atom)
```

Return the current active entry label.

add

```
add(+E)
```

Add a new entry. The entry must have the form: `ColorAtom#LabelAtom`.

rem

```
rem(+LabelA)
```

Remove (all) entries with the label +LabelA.

28.5 Textframe

The `TkTools.textframe` class extends `Tk.frame` and has only one method.

tkInit

```
tkInit(parent:  +WidgetO
       'class': +ClassTcl <= unit
       text:    +TextTcl
       font:    +FontO    <= DefaultFont)
```

Initialise the widget object and create a new textframe.

28.6 Notebook

The `TkTools.notebook` class extends `Tk.canvas`.

tkInit

```
tkInit(parent:  +WidgetO
       font:    +FontO    <= DefaultFont)
```

Initialise the widget object and create a new notebook.

add

```
add(+NoteO)
```

Add a new note.

remove

```
remove(+NoteO)
```

Remove a note.

toTop

```
toTop(+NoteO)
```

Bring note +NoteO to the top.

getTop

```
getTop(?NoteO)
```

Return the top note.

28.7 Note

The TkTools.note class extends Tk.frame.

tkInit

```
tkInit(parent:  +WidgetO
       text:    +TextTcl)
```

Initialise the widget object and create a new note. The note packs itself, thus you should avoid sending packaging commands, otherwise the system hangs.

toTop

```
toTop()
```

This is an empty method. If yo want any action to be performed you have to derive the TkTools.note class and to overwrite the toTop method.

28.8 Scale

The TkTools.scale class extends Tk.frame.

init

```
init(parent:  +WidgetO
     width:   +WidthI
     values:  +ValuesL
     initpos: +PosI)
```

Initialise the widget object and create a new scale.The field +ValuesL describes all possible values, whereby the initial value is described by position +PosI.

drawTicks

```
drawTicks()
```

Draw.

get

```
get(?ValueI)
```

Get the current value.

28.9 Number Entry

The TkTools.numberentry class extends Tk.frame.

tkInit

```
tkInit(parent:       +WidgetO
       min:          +MinI    <= unit
       max:          +MaxI    <= unit
       val:          +ValI    <= MinI
       font:         +FontO   <= DefaultFont
       width:        +WidthI  <= 6
       action:       +ActionP <= DummyAction
       returnaction: +RetActP <= unit)
```

Initialise the widget object and create a new number entry. The number entry contains all values within +MinI and +MaxI, and shows initially the value +ValI. Allthough it is possible to set +ValI less than +MinI you should not do it.

By +ActionP you define an unary procedure that is each time invoked, when alter the value of the number entry. Furthermore you may set a return procedure, that is invoked when you press Return.

tkAction

```
tkAction(+ActionP <= unit)
```

Set a new action procedure.

tkSet

```
tkSet(+ValI)
```

Set a new value.

tkGet

```
tkGet(?ValI)
```

Get value.

enter

```
enter(+B)
```

Hm. I see no difference.

28.10 Images

The function

```
{TkTools.images +L ?+R}
```

takes a list of URLs as input and returns a record of images, where the fields are atoms derived naturally from the URLs.

First the basename of the URL is computed by taking the last fragment of the URL (that is, 'wp.gif' for example). The extension (the part following the period, 'gif' for example), determines the type and format of the image. The part of the basename that precedes the period yields the feature.

28.11 Resolve Images

TkTools.resolveImages is deprecated. Please use TkTools.images instead.

Part VII

Miscellaneous

Support Classes for Objects: ObjectSupport

This module contains classes that provide generic functionality for objects: Organizing objects in hierarchies and reflection of objects.

29.1 Classes for Master/Slave Behaviour

An instance `MasterO` of class `Object.master` becomes a master of an `Object.slave` object when the latter receives the message `becomeSlave(MasterO)`.

Methods for Masters

`init`

`init()`

initialization; mandatory for internal reasons.

`getSlaves`

`getSlaves(?SlaveOs)`

returns the list of current slaves.

An instance of `Object.slave` becomes a slave of an `Object.master` object `MasterO` when it receives the message `becomeSlave(MasterO)`.

Methods for Slaves

`becomeSlave`

`becomeSlave(+MasterO)`

makes **`self`** become a slave of `MasterO`. **`self`** must not yet be a slave of any object, else an exception is raised.

`isFree`

`isFree(?B)`

tests whether **`self`** is not the slave of any object.

`free`

`free()`

frees **`self`**. **`self`** must be the slave of some object, else an exception is raised.

29.2 Reflecting Objects

The unsited class `ObjectSupport.reflect` provides the following methods:

`clone`

`clone(+O)`

returns a clone of **`self`** (that is, features and current attribute values are equal).

`toChunk`

`toChunk(?Ch)`

returns a chunk that contains information on current attribute values and features.

`fromChunk`

`fromChunk(+Ch)`

Sets features and attributes according to chunk `Ch`. `Ch` must have been created with `toChunk`. The current object must have the same class as that from which the chunk was computed.

Bibliography

[1] P. Baptiste, C. Le Pape, and W. Nuijten. Incorporating efficient operations research algorithms in constraint-based scheduling. In *First International Joint Workshop on Artificial Intelligence and Operations Research*, 1995.

[2] Y. Caseau and F. Laburthe. Disjunctive scheduling with task intervals. LIENS Technical Report 95-25, Laboratoire d'Informatique de l'Ecole Normale Superieure, 1995.

[3] Donald Lewine. *POSIX Programmer's Guide*. O'Reilly & Associates, Inc., April 1991.

[4] P. Martin and D.B. Shmoys. A new approach to computing optimal schedules for the job shop scheduling problem. In *International Conference on Integer Programming and Combinatorial Optimization, Vancouver*, pages 389–403, 1996.

[5] Tobias Müller and Martin Müller. Finite set constraints in Oz. In François Bry, Burkhard Freitag, and Dietmar Seipel, editors, *13. Workshop Logische Programmierung*, pages 104–115, Technische Universität München, 17–19 September 1997.

[6] John K. Ousterhout. *Tcl and the Tk Toolkit*. Professional Computing Series. Addison-Wesley, Reading, MA, USA, 1994.

[7] Jean-François Puget. A fast algorithm for the bound consistency of alldiff constraints. In *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI-98)*, pages 359–366, Madison, WI, USA, July 1998. AAAI Press/The MIT Press.

[8] Jean-Charles Régin. A filtering algorithm for constraints of difference in CSPs. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, pages 362–367, Seattle, WA, USA, 1994. AAAI Press.

[9] Christian Schulte. Oz Explorer: A visual constraint programming tool. In Lee Naish, editor, *Proceedings of the Fourteenth International Conference on Logic Programming*, pages 286–300, Leuven, Belgium, July 1997. The MIT Press.

[10] J. Würtz. Oz Scheduler: A workbench for scheduling problems. In M.G. Radle, editor, *Eighth International Conference on Tools with Artificial Intelligence*, pages 149–156, Toulouse, France, 1996. IEEE, IEEE Computer Society Press.

[11] J. Würtz. Constraint-based scheduling in Oz. In U. Zimmermann, U. Derigs, W. Gaul, R. Möhrig, and K.-P. Schuster, editors, *Operations Research Proceedings 1996*, pages 218–223. Springer-Verlag, Berlin, Heidelberg, New York, 1997. Selected Papers of the Symposium on Operations Research (SOR 96), Braunschweig, Germany, September 3–6, 1996.

Index