MLWorks™

Reference Manual

Version 2.0

harlequin™

Copyright and Trademarks

MLWorks™ Reference Manual

July 1998

Part number: MLW-2.0.0-RM

Europe:

Harlequin Limited
Barrington Hall
Barrington
Cambridge CB2 5RG
U.K.

telephone +44 1223 873 800
fax +44 1223 872 519

North America:

Harlequin Incorporated
One Cambridge Center
Cambridge, MA 02142
U.S.A.

telephone +1 617 374 2400
fax +1 617 252 6505

Electronic Access:

http://www.harlequin.co.uk
http://www.harlequin.com

Contents

1

Introducing the MLWorks Libraries

1.1 Introduction

Each of the libraries supplied with MLWorks is described in this reference manual, with the exception of the Standard ML Basis library, for which documentation can be found on the Internet. See the release notes for your version of MLWorks for details of where to find the Basis library documentation.

The libraries documented in this manual are:

- The MLWorks pervasive library
- The MLWorks interactive environment library
- The MLWorks Motif interface library
- The MLWorks Windows interface library
- The MLWorks foreign interface library

We also look at the items provided unqualified at top level in the interactive environment.

1.2 The top level of the MLWorks interactive environment

The MLWorks interactive environment, invoked either as MLWorks, or as MLWorks with the Standard ML Basis library, provides the structures

`MLWorks`, **`General`** and **`Shell`** at top level. A a number of other items unqualified by module names are also available, such as basic ML types and arithmetic operators.

The **`MLWorks`** and **`Shell`** structures are MLWorks libraries discussed later in this chapter. The **`General`** structure is part of the Standard ML Basis library.

See Chapter 2, "The MLWorks Interactive Environment Top Level", for more details of the top level.

1.3 The MLWorks pervasive library

The MLWorks pervasive library is available in interactive environment in the built-in structure **`MLWorks`**. It is also available separately in the MLWorks installation as a set of source and compiled files so that you can use it in your applications.

The pervasive library is MLWorks' own general-purpose library containing facilities for I/O, creation of standalone applications, multiprocessing, profiling, and so on. There is some overlap between the facilities provided by the pervasive library and those provided by the Standard ML Basis library.

See Chapter 3, "The MLWorks Pervasive Library", for more details.

1.4 The MLWorks interactive environment library

The MLWorks interactive environment library is available in the interactive environment in the built-in structure **`Shell`**. It is not available separately, as a set of source and compiled files, because it is only relevant to the interactive environment.

The interactive environment library provides a number of facilities for programmatically customizing the behavior of the MLWorks interactive environment.

See Chapter 4, "The MLWorks Interactive Environment Library", for more details.

1.5 The MLWorks Motif interface library

The MLWorks Motif interface library is not loaded into the interactive environment. It is supplied as a set of source and compiled files on disk, suitable for loading into the interactive environment or for building into your applications.

The Motif interface library provides an ML interface to the X Window System with Motif, including relevant parts of the X Toolkit, and to the graphics functionality of Xlib.

See Chapter 5, “The MLWorks Motif Interface Library”, for more details.

1.6 The MLWorks Windows interface library

The MLWorks Windows interface library is not available in the interactive environment. It is supplied as a set of source and compiled files on disk, suitable for loading into the interactive environment or for building into your applications.

The Windows interface library provides an ML interface to a selection of Windows SDK functions concerned with window-programming tasks, to allow you to write windowing applications in MLWorks.

See Chapter 6, “The MLWorks Windows Interface Library” for more details.

1.7 The MLWorks foreign interface library

The MLWorks foreign interface library (FI, for short) is is not built in to the MLWorks interactive environment, but is provided as a set of source and separately compiled files. See the installation notes for your version of MLWorks for details of their location.

The MLWorks foreign interface library provides facilities for interfacing ML applications to code written in C. The design accommodates other languages, and we may add support for them in future.

See Chapter 7, “The MLWorks Foreign Interface Library” for more details.

2

The MLWorks Interactive Environment Top Level

2.1 Introduction

This chapter describes the items available at top level in the MLWorks interactive environment, as invoked with **`mlworks`** or **`mlworks-basis`**. These are:

- The Standard ML Basis Library
- The structures **`MLWorks`**, **`General`**, and **`Shell`**
- A number of identifiers unqualified by module names

2.2 The MLWorks and Shell structures

The **`MLWorks`** structure is documented in Chapter 3, "The MLWorks Pervasive Library". The **`Shell`** structure is documented in Chapter 4, "The MLWorks Interactive Environment Library".

2.3 The Standard ML Basis library

The **`General`** structure is part of the Standard ML Basis library, which is not documented here. See the release notes supplied with your version of MLWorks for details of online documentation available for the Standard ML Basis library. The *MLWorks User Guide* also contains further information on the Basis library.

2.4 Unqualified top-level items

Every item available unqualified at top level is as defined by Standard ML Basis library, except for the function **`use`**, which is defined only in the interactive environment.

The items include types, arithmetic operators, and utility functions. Some of the items are primitives of the top level, while others are defined by Basis library modules such as **`General`**, **`Int`**, **`Word`**, **`Real`**, **`Char`**, **`String`**, **`Substring`**, **`Array`**, and **`Vector`**.

The Standard ML Basis library is not documented here. See the release notes supplied with your version of MLWorks for details of documentation available for the Standard ML Basis library.

The **`use`** function is described below.

use *Function*

Summary: Reads ML declarations from a file and evaluates them.

Type: **`val use : string -> unit`**

Syntax: **`use`** ***file*** **`-> ()`**

Arguments: *file* — A string naming a file containing ML declarations.

Values: **`unit`**

Description: Reads ML declarations from a single *file* and evaluates them, thereby creating new bindings in the interactive environment. The *file* string should contain an ordinary filename specified in the filename notation of the underlying operating system. It should not use the MLWorks project system's abstract unit and compound notation. Nor should *file* contain top-level **`require`** declarations, though **`use`** declarations are permitted.

3

The MLWorks Pervasive Library

3.1 Introduction

The pervasive library is built in to MLWorks, in the permanently available structure **`MLWorks`**. It provides a variety of MLWorks-specific features. These include a concurrency ("threads") mechanism and a programmatic interface to the MLWorks profiler.

The pervasive library is also available as a set of separately compiled and source files. See the installation notes for your version of MLWorks for details of their location.

The **`MLWorks`** structure has the following skeletal signature:

```
signature MLWORKS =
 sig
   exception Interrupt
   val arguments: unit -> string list
   structure Deliver
   structure Internal
   structure Profile
   structure String
   structure Threads
 end
```

3.1.1 Supported features

The supported items, are, then:

- The top-level items **`Interrupt`** (an exception) and **`arguments`** (a function).
- The **`Deliver`** structure, which provides application delivery tools.
- The **`Profile`** structure, which provides a programmatic interface to the MLWorks profiler.
- The **`Threads`** structure, which provides multiprocessing features.

3.2 Top-level items: Interrupt and arguments

The exception **`Interrupt`** and the function **`arguments`** are defined at the top level of the **`MLWorks`** structure.

Interrupt *Exception*

Summary	Exception that MLWorks raises for fatal signals, stack overflows, and other interrupt events.
Structure	**`MLWorks`**
Type	**`exception Interrupt`**
Description	The **`Interrupt`** exception is raised following an interrupt, a fatal signal, a stack overflow, or a breakpoint when running the MLWorks GUI environment. When one of these events occurs, MLWorks enters the stack browser. If you abort the stack browser, **`MLWorks.Interrupt`** is raised.

arguments *Function*

Summary
: Returns the command line arguments with which the current image was called.

Structure
: **`MLWorks`**

Syntax
: **`arguments () ->`** ***arguments***

Description
: Returns the command line arguments with which the current image was called.

3.3 Delivery tools: the Deliver structure

Delivery is the process of creating an image file or standalone executable file from an ML function. The **`Deliver`** structure provides tools for performing delivery. The structure has the following signature:

```
structure Deliver :
sig
  type deliverer = string * (unit -> unit) * bool -> unit
  type delivery_hook = deliverer -> deliverer
  val deliver : deliverer
  val with_delivery_hook : delivery_hook -> ('a -> 'b) -> 'a -> 'b
end
```

The components of the **`Deliver`** structure are described below.

deliverer *Type*

Summary
: The type of a delivery function.

Structure
: **`MLWorks.Deliver`**

Type
: **`type deliverer = string * (unit -> unit) * bool -> unit`**

Description
: The type of a function that takes a string, a function of type **`unit -> unit`**, and a boolean. This is the

type of the **deliver** function, which implements the delivery process. See **deliver**, page 18.

delivery_hook *Type*

Summary
The type of a function that, given a delivery function, augments it to return another.

Structure
MLWorks.Deliver

Type
type delivery_hook = deliverer -> deliverer

Description
The type of a function that, given a delivery function, augments it to return another. This allows you to customize the delivery process.

deliver *Function*

Summary
Delivers an ML function into an image file or a standalone executable file.

Structure
MLWorks.Deliver

Syntax
deliver ***file function executable?*** **-> ()**

Arguments

file	A string naming a file into which to write the image or executable.
function	An ML function to deliver. It must be of type **unit -> unit**.
executable?	A boolean value. If **true**, *file* will be a standalone executable; if false, *file* will be an image file.

Values
The **deliver** function returns **unit**.

Description

Delivers an ML function into an image file or a standalone executable file.

If *standalone?* is **`true`**, *file* can be executed standalone. If **`false`**, *file* can be executed by passing it to the MLWorks runtime.

When **`deliver`** is called on a function, MLWorks performs a garbage collection to remove as much irrelevant code as possible. After garbage collection, MLWorks writes the delivered *function* to *file*.

Note: Under Windows, MLWorks exits once delivery is complete, whereas under UNIX, it continues to operate.

Example

Consider the following function:

```
fun hello() = print "Hello, world.\n";
```

To deliver a standalone executable version of it, call:

```
deliver ("hello", hello, true);
```

Then to run it, on the OS command line, call:

```
> hello
Hello, world.
>
```

To deliver an image file of it, call:

```
deliver ("hello.img", hello, false);
```

Then to run it, go to the OS command line. On Windows, produce an MS-DOS prompt and call:

```
dos> bin\mlimage-console hello.img
Hello, world.
dos>
```

On UNIX, call:

```
unix> bin/mlimage hello.img
Hello, world.
unix>
```

with_delivery_hook *Function*

Summary Adds a user-customized delivery function to a list of hooks that will be executed during delivery.

Structure `MLWorks.Deliver`

Syntax `with_delivery_hook` ***`hook f x -> fn`***

Arguments

hook	A user-customized delivery function of type `delivery_hook`.
f	An ML function.
x	Arguments to *f*.

Values

fn	A function.

Description Adds a user-customized delivery function, *hook*, to a list of hooks that will be executed during delivery whenever the `deliver` function is called *during the application* of *f* to *x*. Note that hooks are executed in the order they were added: if *h1* is added first, it will be executed first.

Example The following example defines a hook that catches a failed delivery:

```
MLWorks> fun hello () = print "Hello,
world.\n";
val hello : unit -> unit = fn
```

```
MLWorks> fun deliver_hello () =
  MLWorks.Deliver.deliver("hello", hello,
true);
val deliver_hello : unit -> unit = fn
```

```
MLWorks> MLWorks.Deliver.with_delivery_hook
(fn f => fn
  x => f x handle _ => print "Delivery
failed\n")   deliver_hello ();
val it : unit = ()
MLWorks>
```

3.4 Profiling: the Profile structure

The **`Profile`** structure provides a programmatic interface to the MLWorks profiler. Because this programmatic interface is richer than that provided by the GUI environment's profiler tool, you may want to use it in preference to that tool.

There are three kinds of profiling available: time profiling, space profiling, and call-counting. You can also specify the frequency with which the profiler should scan the stack, in milliseconds.

The profiler supports the notion of a profiling *manner*, which is a convenient way of specifying the kind of information the profiler should gather. A manner might specify that only call-counting should be done, and that the number of garbage collections that occurred during the execution of the function should be recorded.

When the function call is complete, the profiler returns the results of the call and the results of profiling.

Note: The profiler interface contains some details pertaining to cost-centre profiling. That style of profiling may be implemented in a future release.

The following sections explain the use of the profiler in more detail. The example code here does not qualify identifiers with an **`MLWorks.Profile`** prefix for the sake of brevity; you should add this prefix yourself, or use **`local open MLWorks.Profile in`** ... **`end`**, in order to run these examples.

3.4.1 Code items

The profiler produces execution profiles for all *code items*. A code item is an individual piece of machine code generated by the compiler, so is the atomic unit of code as far as the interactive MLWorks environment is concerned. We shall for convenience's sake talk about profiling functions or function calls, but it is worth looking into code items to understand the profiler better.

A code item most closely corresponds to an instance of a lambda function (a **`fn`**) in the source. So in:

```
val f = fn x => x + 10;
```

The **`fn x => x+10`** code item is profilable: an application of **`f`** to an integer value could be profiled, though in practice its execution would probably be completed so quickly that the profiler would not have time to record anything about the application.

Consider the functions **`foo`** and **`bar`**:

```
fun foo x = x + 10;
val bar = foo;
```

In calls to **`foo`** and **`bar`**, the function **`foo`** would be profiled (it has an implicit lambda) but **`bar`** would not be profiled separately from **`foo`** because it shares a code item with it.

```
fun frob x y = x + y + 10;
val bar = frob;
val baz = frob 3;
val qux = baz;
```

In the functions defined above, there are two code items for **`frob`**, one which is invoked when **`frob`** is applied to one argument, and one which is invoked when the result is further applied to another argument: **`frob`** has two implicit lambdas. These will appear separately in a profile. The functions **`bar`**, **`baz`** and **`qux`** are really just identifiers, not new code items, so will not show up separately in the profiler.

The compiler might discard code items by inlining them at their call sites. It might also generate new code items when optimizing function-calling sequences. For instance, a third code item may be generated for the function **`frob`** above, which is used when **`frob`** is called with two known arguments in machine registers.

Every code item has a name. Code items from regular functions inherit the function name. Code items from explicit lambdas (as in the first example above), are called **`<anon>`**. Code items from curried functions (like **`frob`**) have

argument 1 and **argument 2** appended to their names. Code items generated when optimizing have text such as **<Entry>** appended to their names.

3.4.2 Invoking the profiler

To profile the evaluation of a function call, you use the **profile** function. The form of a call to **profile** is:

profile ***options f x***

This call produces an execution profile of the evaluation of the function *f* applied to the value *x*. Arguments are passed to the profiler with *options*, a value that specifies a stack-scanning interval (in "user" milliseconds, the time that is devoted solely to evaluating user code rather than internal MLWorks operations) and a selector function that determines a profiling manner to be used when profiling *f*.

3.4.3 Profiling manners

Profiling manners, which describe how the profiler should gather profiling data, are represented by the type **manner**. To define a manner, use the function **make_manner**. The function takes a record describing how to perform an execution profile, and returns a **manner**:

```
val make_manner :
  {time : bool,
   space : bool,
   calls : bool,
   copies : bool,
   depth : int,
   breakdown : object_kind list} -> manner
```

By providing values for these fields, you specify a manner in which to profile a function:

time	If **true**, gather time-profiling data.
space	If **true**, gather space-profiling data.
calls	If **true**, gather function-call-counting data.
copies	If **true**, record the number of garbage-collector copies per generation in any space profile.

depth An integer specifying the call-depth to which callers should be recorded in multi-level time profiling. See the note on multi-level time profiling in Section 3.4.6 on page 26.

breakdown A list of the kinds of object to break down in any space profile. Of the total amount of space allocated by a function, breakdowns show how much of that space was taken by a certain kind of object. See **object_kind**, page 40.

3.4.4 Specifying scan frequency and profiling manner details

As we have seen, the **profile** function takes an argument *options*, which is a value of type **options**:

```
datatype options =
  Options of
  {scan : int,
   selector : function_id -> manner}
```

The fields are:

scan The interval in "user" milliseconds between scans of the stack. If **scan** is 0, no scans will occur. (By "user" milliseconds, we mean milliseconds of execution time that are devoted solely to running your program, rather than on MLWorks internals.)

The profiler may not be able to respect very small values for **scan**, because it relies on the underlying operating system clock. See **options**, page 33, for details of choosing a realistic value for **scan**.

selector A function that determines the manner that will be used when profiling a particular function. When the profiler is invoked, the **selector** function is applied to every function in the MLWorks heap — including those of MLWorks internals.

The **selector** function should take a value of type **function_id**. Function IDs are strings that begin with the name of the function they identify, but that also contain source and, possibly, compilation information. MLWorks generates a function ID for every function it compiles; you do not have to generate them yourself. The **selector** function should return a value of type **manner**.

An example best explains how manners and selectors are used. Suppose that an application contains a set of functions for writing data to disk. The functions write data in differing quantities: **write_element**, **write_line**, and **write_record**. Being involved in disk I/O, these functions spend a lot of time in execution. On the other hand, they are not called very often. You might construct a specific profiling manner for these functions, **write_fns_manner**, which ensures that time-profiling and space-profiling data is collected for **write** functions, but that call counts are not.

To construct the manner:

```
val write_fns_manner = make_manner {
  time = true,
  space = true,
  calls = false,
  copies = true,
  depth = 1,
  breakdown = [ TOTAL ]
};
```

See **make_manner**, page 29, for more details. A suitable selector function might be:

```
local
  fun is_write_fn s = String.size s >= 5
    andalso String.substring (s,0,5) = "write"
in
  fun my_selector s = if is_write_fns s then write_fns_manner
                      else generic_manner
end;
```

This selector returns **write_fns_manner** for profiling any function whose identifier has the prefix **write**. Any other function will be profiled with the manner **generic_manner**.

Now you can construct a set of options to pass to **profile**. A **scan** value of 10 (100 clock ticks per second) or so is typical on both UNIX and Windows.

```
val my_options = Options { scan = 10, selector = my_selector };
```

3.4.5 Profiling results

When the evaluation of *f x* is complete, the profiler returns both the result of *f x*, and the execution profile that was constructed for it.

These results are expressed in a pair of type **'a result * profile**. The **profile** is the most interesting here, as it is a record containing several values. One of the values is of type **function_profile**, which is the type of function execution profiles. The other values (of types **call_header**, **general_header**, **space_header**, **time_header**) record information about profiler performance and are not generally of interest.

The execution profile for the function itself is returned in the value of type **function_profile** in the **profile** value's **functions** field. The **function_profile** value contains four fields:

id	A string containing the profiled function's name, its location, and, possibly, compilation information.
call_count	The total number of times the profiled function was called.
time	A value of type **function_time_profile** containing time-profiling results. This field will have no useful value unless you request time profiling in the profiling manner.
space	A value of type **function_space_profile** containing space-profiling results. This field will have no useful value unless you request space profiling in the profiling manner.

3.4.6 Time profiling

The profiler offers time profiling. You can request it by setting the **time** field to **true** when creating a manner with **make_manner**.

In time profiling, you specify a stack-scanning interval (in "user" milliseconds) via the **options** type. For each function you time-profile, the profiler returns a value of type **function_time_profile** in the **time** slot of the **function_profile**. The contents of the **function_time_profile** record are only meaningful if the profiling manner used requested time-profiling information.

There are two kinds of time profiling: *single-level* time profiling and *multi-level* time profiling. Single-level time profiling only gathers data about a single function. Multi-level time profiling allows frequency information to be gathered on "**`foo`** called **`bar`** called **`baz`**" patterns, to a depth specifiable in the **`depth`** field of the record you pass to **`make_manner`**. This information is reported by generating a tree of "*x* called *y*" branches, each with statistical information attached.

The values in the fields of the **`function_time_profile`** are:

- The total number of times the function was found on the stack (**`found`**).
- The total number of times the function was found on top of the stack (**`top`**).
- The total number of scans in which the function was found at least once (**`scans`**).
- The maximum number of times that the function was found on the stack in a single scan (**`depth`**).
- The maximum number of self-recursions the function performed in a single scan (**`self`**).
- A list of the functions that called the function (**`callers`**). This field is meaningful only in a multi-level time profile.

3.4.7 Space profiling

The profiler offers space profiling. You can request it by setting the **`space`** field to **`true`** when creating a manner with **`make_manner`**. For each function you space-profile, the profiler returns a value of type **`function_space_profile`** in the **`space`** field of the **`function_profile`**. The contents of the **`function_space_profile`** record are only meaningful if the profiling manner used requested space-profiling information.

The following data is recorded in the fields of the **`function_space_profile`**:

- The total amount of data allocated by the function during its execution (**`allocated`**).

- The total amount of the allocated data that the garbage collector copied (**copied**).
- A breakdown of **copied**, by garbage-collector generation number (**copies**).
- A breakdown of **allocated**, by object kind (**allocation**). Of the total amount of space allocated by a function, breakdowns show how much of that space was taken by a certain kind of MLWorks runtime object. See **object_kind**, page 40.

3.4.8 Profiler performance data

In addition to data gathered for your function according to the profiling manner used, the profiler records data about its own performance. Values of type **call_header**, **space_header**, **time_header** and **general_header** are returned in every **profile**. See the reference entries for these types (pages 42 to 45) for more details.

3.4.9 Profile structure values

profile *Function*

Summary	Produces an execution profile of the evaluation of a single function.	
Structure	**MLWorks.Profile**	
Type	**val profile : options -> ('a -> 'b) -> 'a -> ('b result * profile)**	
Syntax	**profile *options f x* -> (*result*, *profile*)**	
Arguments	*options*	Profiler options. A value of type **options**.
	f	A function to profile.

	x	An argument to *f*.
Values	*result*	The result of the evaluation of *f x*. A value of type **'a result**.
	profile	Profile data for *f x*. A value of type **profile**.

Description

Produces an execution profile of a single function *f* applied to the value *x*. Profile data is returned in a value of type **profile**.The result is a value of type **'a result**.

The profile type contains profiler performance data and an execution profile for *f*. See the entry for **profile**, page 31, for more details.

See also

function_profile, page 34

profile, page 31

call_header, page 42

space_header, page 42

time_header, page 43

general_header, page 45

make_manner *Function*

Summary

Creates a profiling manner.

Signature

MLWORKS.Profile

Structure

MLWorks.Profile

Type

```
val make_manner :
  {time : bool,
   space : bool,
   calls : bool,
   copies : bool,
   depth : int,
   breakdown : object_kind list} -> manner
```

Syntax

make_manner ***manner_spec*** **->** ***manner***

Description

Creates a profiling manner.

time	If **true**, return time-profiling data.
space	If **true**, return space-profiling data.
calls	If **true**, return function-call-counting data
copies	If **true**, record the number of garbage-collector copies per generation in any space profile.
depth	An integer specifying the call-depth to which callers should be recorded in multi-level time profiling. See the note on multi-level time profiling in Section 3.4.6 on page 26.
breakdown	A list of the kinds of MLWorks runtime object to break down in any space profile. Of the total amount of space allocated by a function, breakdowns show how much of that space was taken by a certain kind of object. See **object_kind**, page 40.

profile *Type*

Summary The type used to record profiler results.

Structure `MLWorks.Profile`

Type

```
datatype profile = Profile of
  {general: general_header,
   call: call_header,
   time: time_header,
   space: space_header,
   cost: cost_header,
   functions: function_profile list,
   centres: cost_centre_profile list}
```

Description The type used to record profiler results. The **profile** function returns a value of this type. The record contains the following data:

functions A value of type function_profile list. This value contains the execution profile of each function call profiled.

general A value of type **general_header**. This value contains general data about the allocation during and time taken by the call to the profiler.

call A value of type **call_header**. This value records the number of functions that were call-count profiled.

time A value of type **time_header**. This value records overall time-profiling statistics.

space A value of type **space_header**. This value records overall space-profiling statistics.

cost A value of type **cost_header**. This value is unused at present. It may record cost-centre profiling information in a future release.

centres A value of type cost_centre_profile. This value is unused at present. It may record cost-centre profiling information in a future release.

See also

call_header, page 42

function_profile, page 34

general_header, page 45

space_header, page 42

time_header, page 43

'a result *Type*

Summary

The type of the result returned from a profiled function.

Structure

MLWorks.Profile

Type

```
datatype 'a result =
    Result of 'a
  | Exception of exn
```

Description

The type of the result returned from a profiled function. The result in question is the normal result of evaluating the function call *f x*, not the execution profile itself.

Either the function returns a value, in which case the profiler returns **Result** *value*, or it raises an exception, in which case the profiler returns **Exception exn**.

manner *Type*

Summary The type of a profiling manner.

Structure `MLWorks.Profile`

Type

```
type manner
```

Description The type of a profiling manner.

A profiling manner describes how the profiler should gather profiling data. To define a manner, use the function **make_manner**.

See also **make_manner**, page 29

options *Type*

Summary Profiler options.

Structure `MLWorks.Profile`

Type

```
datatype options =
  Options of
    {scan : int,
    selector : function_id -> manner}
```

Description This type is used for passing options to the profiler. The profiler needs to know how often to scan the stack. It also needs a selector function, which is used to select a profiling manner with which to profile the function call passed to **MLWorks.Profile.profile** as *f x*.

scan The interval in “user” milliseconds between scans of the stack. If 0 then no scans will occur.

By "user" milliseconds, we mean the time that is devoted solely to evaluating user code, rather than internal MLWorks operations.

The profiler may not be able to respect very small values for **scan**, because it relies on the underlying operating system clock. Most UNIX operating systems provide 100 ticks per second (**scan** $\geq$ 10 realistic) or 60 ticks per second (**scan** $\geq$ 16 realistic). The clock granularity on Windows can vary from machine to machine, but **scan** values similar to those for UNIX are usually fine.

selector Selects the manner in which particular functions are to be profiled.

function_profile *Type*

Summary
Records all information gathered during the profiling of a function.

Structure
MLWorks.Profile

Type

```
datatype function_profile = Function_Profile of
  {id: function_id,
   call_count: int,
   time: function_time_profile,
   space: function_space_profile}
```

Description
This datatype records all information gathered during the profiling of a function. The record contains the following data:

`id`	A string containing the profiled function's name, its location, and, possibly, compilation information.
`call_count`	The total number of times the profiled function was called.
`time`	A value of type **`function_time_profile`** containing time-profiling results. This field will have no useful value unless you request time profiling in the profiling manner.
`space`	A value of type **`function_space_profile`** containing space-profiling results. This field will have no useful value unless you request space profiling in the profiling manner.

function_time_profile *Type*

Summary

Records time-profiling information gathered for a particular function.

Structure

`MLWorks.Profile`

Type

```
datatype function_time_profile =
  Function_Time_Profile of
    {found: int,
     top: int,
     scans: int,
     depth: int,
     self: int,
     callers: function_caller list}
```

Description

This datatype records time-profiling information about a function. If you ask the profiler to time-profile a particular function, the results of profiling will be stored in a **function_time_profile** value in the **time** field of the **function_profile** returned by the profiler. See **make_manner** for a discussion of how to ask for time profiling.

The record contains the following data:

found	The number of times that the function was found on the stack. This will be -1 if time profiling was not requested in the profiling manner used.
top	The number of times that the function was found on the top of the stack.
scans	The number of scans in which the function was found on the stack at least once.
depth	The maximum number of recursions performed in a single scan.
self	The maximum number of self-recursions performed in a single scan.
callers	A list of the functions that called the profiled function, with timing data for those functions attached.

Roughly, **top** indicates how much time was spent in *f* alone, and **scans** how much time was spent in *f* and all the functions it called. The average recursion of the function is roughly **found** divided by **scans**, and **depth** and **self** are indicators of the maximum recursion of the function. These are all rough indicators because of the random element introduced by time-based sampling and because optimizations such as tail-call removal have an effect on the stack.

Example

Suppose the function *f* was found in two scans. On the first occasion the stack looked like this (the top of the stack is listed first):

- *f,f,f,f,g,f,g,f,g,f,g,f,g,f,g,f*

On the second occasion it looked like this (the top is again listed first):

- *g,g,g,g,f,g,f,g,f,g,f,f,g,g,f,f*

Then:

`found` would be 17.

`top` would be 1. (From the first scan.)

`scans` would be 2.

`depth` would be 9. (From the first scan.)

`self` would be 3. (From the first scan.)

See also

`function_caller`, page 37

`function_profile`, page 34

function_caller *Type*

Summary

Records profile information about the caller of a particular function.

Structure

`MLWorks.Profile`

Type

```
datatype function_caller = Function_Caller of
  {id: function_id,
   found: int,
   top: int,
   scans: int,
   callers: function_caller list}
```

Description

This datatype records the profile information about the caller of a particular function in multi-level time profiling. The record contains the following data:

id	The identifier of the function.
found	The total number of times that the function was found on the stack.
top	The number of times that the function was found on top of the stack.
scans	The number of stack scans in which the function was found.
callers	A list of the functions that called the function.

See also

function_time_profile, page 35

function_space_profile *Type*

Summary

Records space-profiling information gathered for a particular function.

Structure

MLWorks.Profile

Type

```
datatype function_space_profile =
  Function_Space_Profile of
   {allocated : large_size,
   copied : large_size,
   copies : large_size list,
   allocation : (object_kind * object_count)
list list}
```

Description

Records space-profiling information gathered for a particular function. The record contains the following data:

`allocated` Total amount of data allocated by the function during its execution. This will be -1 if space profiling was not requested in the profiling manner used.

`copied` Total amount of data copied by the garbage collector that was allocated by the profiled function.

`copies` A breakdown of **`copied`** by generation number: **`copies`** is a list **`[`***s*1**`,`** *s*2**`,`** *s*3**`,`** …**`]`** where *s*1 is the amount of data copied out of generation zero, and so on.

`allocation` A list of breakdowns of data allocated and copied: **`allocation`** is a list **`[`***b*0**`,`** *b*1**`,`** *b*2**`,`** …**`]`** where each *b* is a breakdown, that is, a list **`[(`***k*1**`,`** *c*1**`),`** **`(`***k*2**`,`** *c*2), ...**`]`**. The *k*'s are the **`object_kind`** values in the breakdown list given to **`make_manner`**. Each *c* is the object count corresponding to that runtime object kind, a value of type **`object_count`**. The count gives the number of objects, the size of the data in the objects, and the space overhead (padding, headers, and so on) of the objects.

Each *b* corresponds to a generation: *b*0 is a breakdown of the objects allocated by that function, *b*1 is a breakdown of the runtime objects allocated by that function which the garbage collector copies out of generation zero, *b*2 the breakdown for copies out of generation one, and so on.

Note: If the profiled function allocates only long-lived objects, they will be copied several times. In that case, **`copied`** will be greater than **`allocated`**. Conversely, if the function allocates only transient data, it will not be copied, and so **`allocated`** will be greater than **`copied`**.

object_kind *Type*

Summary
The various kinds of MLWorks runtime object that can be broken down in a function space profile.

Structure
`MLWorks.Profile`

Type
```
datatype object_kind =
   RECORD
 | PAIR
 | CLOSURE
 | STRING
 | ARRAY
 | BYTEARRAY
 | OTHER
 | TOTAL
```

Description
The various kinds of MLWorks runtime object that can be broken down in a function space profile.

Object kinds to be broken down are specified in the **`breakdown`** field of the record you pass to **`make_manner`**. Breakdowns are given in the **`allocated`** field of the value of **`function_space_profile`** returned after function-space profiling.

Of the total amount of space allocated by a function, a breakdown shows how much of that space was taken by a certain kind of object.

The kinds of objects are mostly self evident, but **`OTHER`** is shorthand for weak arrays and code objects; **`TOTAL`** is not an object kind, but for use in the profiling manner. If you specify **`TOTAL`**, you get back a breakdown list with **`TOTAL`** records, which contain the relevant total information for everything allocated by that function.

See also

`function_space_profile`, page 38

`object_count`, page 41

object_count *Type*

Summary

Records allocation and other space statistics for a data-object kind.

Structure

`MLWorks.Profile`

Type

```
datatype object_count =
  Object_Count of
    {number : int,
     size : large_size,
     overhead : int}
```

Description

Records allocation data and other space statistics for a runtime object kind. It is used in space profiling.

`number`	Total number of runtime objects allocated.
size	Total amount of space used.
`overhead`	Total number of bytes used in overheads: header word, padding, and so on.

See also

function_space_profile, page 38

object_kind, page 40

large_size *Type*

Summary

A type for conveniently representing large space counts.

Structure

MLWorks.Profile

Type

```
datatype large_size =
  Large_Size of
    {megabytes : int,
     bytes : int}
```

Description

The **large_size** type provides a convenient representation for large space counts.

call_header *Type*

Summary

The total number of functions call-count profiled.

Structure

MLWorks.Profile

Type

```
datatype call_header = Call of {functions :
int}
```

Description

The total number of functions call-count profiled.

space_header *Type*

Summary

General space-profiling statistics for the profiler invocation.

Structure **MLWorks.Profile**

Type

```
datatype space_header = Space of
  {data_allocated: int,
   functions: int,
   collections: int,
   total_profiled : function_space_profile}
```

Description General space-profiling statistics for the profiler invocation. These statistics are gathered regardless of whether the profiling manner in use gathered space-profiling information for its **function_profile**.

data_allocated Number of bytes allocated by the space profiler.

functions Number of code vectors space-profiled.

collections Number of garbage collections during the profiler invocation.

total_profiled Pointwise total of all **function_space_profiles**. This will be zero if space-profiling was not requested for any function.

time_header *Type*

Summary General time-profiling statistics for the profiler invocation.

Structure **MLWorks.Profile**

Type

```
datatype time_header = Time of
  {data_allocated: int,
   functions: int,
   scans: int,
   gc_ticks: int,
   profile_ticks: int,
   frames: real,
   ml_frames: real,
   max_ml_stack_depth: int}
```

Description

General time-profiling statistics for the profiler invocation. These statistics are gathered regardless of whether the profiling manner in use gathered time-profiling information for its **`function_profile`**.

`data_allocated` Number of bytes allocated by the time profiler.

`functions` Number of code vectors time profiled.

`scans` Total number of stack scans. Will be zero if scanning is off.

`gc_ticks` Number of scans skipped because the system was collecting garbage.

`profile_ticks` Number of scans skipped because the system was in profile code.

`frames` Number of stack frames scanned in total. Will be zero if scanning is off.

`ml_frames` Number of ML frames scanned in total.

`max_ml_stack_depth`

Maximum depth of ML frames found in a single scan.

general_header *Type*

Summary General statistics for the profiler invocation.

Structure **MLWorks.Profile**

Type

```
datatype general_header = General of
  {data_allocated: int,
   period: Time.Interval.T,
   suspended: Time.Interval.T}
```

Description General statistics for the profiler invocation.

data_allocated Number of bytes allocated by general profiling overhead.

period Execution time.

suspended Currently unused.

ProfileError *Exception*

Summary The profiler's exception.

Signature **MLWORKS.Profile**

Structure **MLWorks.Profile**

Type **exception ProfileError of string**

Description The profiler's exception.

3.5 Threads: the Threads structure

The MLWorks threads interface is provided by the structure **MLWorks.Threads**. Conceptually, the MLWorks thread mechanism consists of a set of concurrently evaluating expression closures, or "threads", that share memory.

All threads are assigned a unique identifier, of type **`thread_id`**, and a thread number. All threads have a status, represented with the **`'a result`** datatype.

Warning: You should rely only on thread identifiers for uniquely identifying threads; thread numbers are not important. Thread numbers are handed out sequentially as threads are created, and are provided as a simple reference for, say, application-debugging purposes. Thread numbers are not necessarily unique: after a sufficiently large number, they may wrap around and start again.

At startup, MLWorks consists of two threads: a master thread, and a single evaluation thread. When running in TTY or GUI mode, this latter thread is linked to the shell interface; otherwise, it executes a prespecified function (for example, a function specified by **`MLWorks.Deliver.deliver`** above).

The threads are run in a scheduler loop. By default, they behave like coroutines. That is, only one thread executes at a time until it either finishes or yields control to another, at which point it sleeps waiting to be woken by a "yield" command. Threads can also be run interleaved, using the **`MLWorks.Threads.Internal.Preemption`** mechanism, which interrupts processes after a user-specified number of milliseconds.

The **`'a result`** datatype is used to record the status of a thread. A thread can be polled for its status with the **`result`** function.

Thread pre-emption mode implements the interleaved execution of threads. You specify a pre-emption interval in milliseconds using the **`set_interval`** function. Each thread is allowed to proceed with its evaluation for this time before being "pre-empted" by the next thread on the scheduling loop.

Warning: If you are interleaving a thread with the interactive GUI shell, any printed output from your thread is liable to be interpreted by the GUI as input to the shell.

Threads *Exception*

Summary The threads mechanism's exception.

Structure **`MLWorks.Threads`**

Type `exception Threads of string`

Description The threads mechanism's exception.

'a thread *Type*

Summary The type of threads.

Structure **`MLWorks.Threads`**

Type `type 'a thread`

Description The type of threads. A value of type *t* **`thread`** is a thread that is evaluating an expression of type *t*.

thread_id *Equality type*

Summary The type of thread identifiers.

Structure **`MLWorks.Threads.Internal`**

Type `eqtype thread_id`

Description The type of thread identifiers. This type is an abstract form of **`'a thread`** which exists to make the thread control functions of **`MLWorks.Threads.Internal`** safely typed.

For example, **`thread_id`** is necessary in order to give the type of the function **`all`**, which returns the list of all threads that currently exist, and which will, in all probability, be evaluating expressions of different types.

'a result *Type*

Summary The type of thread status reports.

Structure `MLWorks.Threads`

Type

```
datatype 'a result =
    Running
  | Waiting
  | Sleeping
  | Result of 'a
  | Exception of exn
  | Died
  | Killed
  | Expired
```

Description The type of thread status reports, as returned by the functions **result** and **Internal.state**. Can be one of:

`Running`	The thread is currently running.
`Waiting`	The thread is currently waiting.
`Sleeping`	The thread is currently sleeping.
`Result of 'a`	The thread has completed, with this result.
`Exception of exn`	The thread exited, with this uncaught exception.
`Died`	The thread died. (For example, because of a bus error.)
`Killed`	The thread has been killed.
`Expired`	The thread no longer exists.

See also **result**, page 49

state, page 55

fork *Function*

Summary Forks a new thread.

Structure **`MLWorks.Threads`**

Syntax **`fork`** ***f x*** **`->`** ***t***

Arguments

f	An ML function.
x	Arguments to *f*.

Values

t	A new thread evaluating *f x*.

Description Forks a new thread which evaluates the application of *f* to *x*.

result *Function*

Summary Polls a thread for its status.

Structure **`MLWorks.Threads`**

Syntax **`result`** ***t*** **`->`** ***result***

Arguments

t	A thread.

Values

result	Result of the thread. A value of type **`'a result`**.

Description Polls the thread *t* for its status. Returns the result of *t* if it has finished evaluating, and a status indicator otherwise.

See also **`'a result`**, page 48

sleep *Function*

Summary	Puts a thread to sleep.
Structure	**`MLWorks.Threads`**
Syntax	**`sleep`** ***t*** **`-> ()`**
Arguments	*t* — A thread.
Values	**`()`**
Description	Puts the thread *t* to sleep. Raises the exception **`Threads`** if *t* is already asleep.

wake *Function*

Summary	Wakes a thread up.
Structure	**`MLWorks.Threads`**
Syntax	**`wake`** ***t*** **`-> ()`**
Arguments	*t* — A thread.
Values	**`()`**
Description	Wakes the thread *t* up, if it was asleep. Raises the exception **`Threads`** if *t* is already awake.

yield *Function*

Summary	Yields control to the next thread in the scheduler loop.

Structure `MLWorks.Threads`

Syntax **`yield () -> ()`**

Arguments **`()`**

Values **`()`**

Description Yields control to the next thread in the scheduler loop.

See also **`yield_to`**, page 55

all *Function*

Summary Returns the list of the identifiers of all threads currently scheduled.

Structure `MLWorks.Threads.Internal`

Syntax **`all () ->`** ***ID-list***

Arguments **`()`**

Values *ID-list* A list of thread identifiers.

Description Returns the list of the identifiers of all threads currently scheduled.

children *Function*

Summary Returns the list of threads that are the children of a thread.

Structure `MLWorks.Threads.Internal`

Syntax	**`children`** ***ID*** **`->`** ***ID-list***	
Arguments	*ID*	A thread identifier.
Values	*ID-list*	A list of thread identifiers.
Description	Returns the list of threads that are the children of *ID*, that is, the threads that were initiated by the thread with **`thread_id`** *ID*.	
See also	**`parent`**, page 54	

get_id *Function*

Summary	Returns a thread's identifier.	
Structure	**`MLWorks.Threads.Internal`**	
Syntax	**`get_id`** ***t*** **`->`** ***ID***	
Arguments	*t*	A thread.
Values	*ID*	A thread identifier.
Description	Returns the identifier of thread *t*.	

get_num *Function*

Structure	**`MLWorks.Threads.Internal`**	
Syntax	**`get_num`** ***ID*** **`->`** ***number***	
Arguments	*ID*	A thread identifier.
Values	*number*	A thread number.

Description — Returns the thread number of the thread *ID*.

id *Function*

Summary — Returns the identifier of the thread that is the current process.

Structure — **`MLWorks.Threads.Internal`**

Syntax — **`id () -> ID`**

Arguments — **`()`**

Values — *ID* — A thread identifier.

Description — Returns the identifier of the thread that is the current process.

kill *Function*

Summary — Kills a thread.

Structure — **`MLWorks.Threads.Internal`**

Syntax — **`kill ID -> ()`**

Arguments — *ID* — A thread identifier.

Values — **`()`**

Description — Kills the thread *ID*.

Warning: It is quite possible to kill MLWorks with **`kill`**. The following call, for example, will do just that:

```
kill (id ());
```

parent *Function*

Summary Returns the identifier of a thread's parent thread.

Structure `MLWorks.Threads.Internal`

Syntax **`parent`** ***ID*** **`->`** ***parent-ID***

Arguments

ID	A thread identifier.

Values

parent-ID	A thread identifier.

Description Returns the identifier of a given thread's parent thread. The identifier *parent-ID* is the identifier of the thread that initiated the thread *ID*.

See also **`children`**, page 51

raise_in *Function*

Summary Raise an exception in a thread.

Structure `MLWorks.Threads.Internal`

Syntax `raise_in (ID,e)`

Arguments

ID	A thread identifier.
e	An exception.

Values `()`

Description Raises exception **`e`** in the thread *ID*.

state *Function*

Summary
: Returns the state of a thread.

Structure
: **`MLWorks.Threads.Internal`**

Syntax
: **`state`** ***ID*** **`->`** ***result***

Arguments
: *ID* A thread identifier.

Values
: *result* A thread state. A value of type **`unit result`**.

Description
: Returns the state of the thread *ID*. See **`'a result`** for details of the possible state values.

See also
: **`'a result`**, page 32

yield_to *Function*

Summary
: Yield control to a thread.

Structure
: **`MLWorks.Threads.Internal`**

Syntax
: **`yield`** ***ID*** **`-> ()`**

Arguments
: *ID* A thread identifier.

Values
: **`()`**

Description
: Yield control to thread *ID*, that is, ignore the normal schedule and jump to thread *ID*.

See also
: **`yield`**, page 50

set_handler *Function*

Summary Sets a fatal signal handler for the current thread.

Structure `MLWorks.Threads.Internal`

Syntax **`set_handler`** ***f*** **`-> ()`**

Arguments *f* A function of type **`int -> unit`**.

Values **`()`**

Description Makes the user-defined function *f* the fatal signal handler for the current thread. When a fatal signal (that is, a SIGSEGV, SIGBUS, or SIGKILL signal) is sent to the current thread, *f* is executed prior to the thread's termination.

The integer argument passed to *f* is the number of the fatal signal.

Note: A thread handler is not inherited by the thread's children. Note also that the **`MLWorks.Threads.Internal.kill`** function does not send a fatal signal.

See also **`reset_fatal_status`**, page 56.

reset_fatal_status *Function*

Summary Removes any fatal signal handlers defined for the current thread.

Structure `MLWorks.Threads.Internal`

Syntax **`reset_fatal_status () -> ()`**

Arguments `()`

Values `()`

Description Removes any fatal signal handlers defined with **`set_handler`** for the current thread. This does not affect the handlers of the current thread's children.

See also **`set_handler`**, page 56

get_interval *Function*

Summary Returns the current pre-emption interval used in thread-interleaving mode.

Structure **`MLWorks.Threads.Internal.Preemption`**

Syntax **`get_interval () ->`** ***interval***

Arguments `()`

Values *interval* An integer.

Description Returns the current pre-emption interval in milliseconds. This interval is the period for which a thread will evaluate in thread-interleaving mode before it is pre-empted by the next thread.

See also **`set_interval`**, page 57

set_interval *Function*

Summary Sets the pre-emption interval used in thread-interleaving mode.

Structure **`MLWorks.Threads.Internal.Preemption`**

Syntax **`set_interval interval -> ()`**

Arguments *interval* An integer.

Values **`()`**

Description Sets the pre-emption *interval*, in milliseconds. The interval is the period for which a thread will evaluate in thread-interleaving mode before it is pre-empted by the next thread.

The default interval value is 0. If the interval is set to 0, the thread mechanism behaves as though interleaving mode were switched off.

See also **`get_interval`**, page 57

`on`, page 58

on *Function*

Summary Returns **`true`** if the threads that make up MLWorks are being interleaved via regular pre-emptions.

Structure **`MLWorks.Threads.Internal.Preemption`**

Syntax **`on () ->`** ***on?***

Arguments **`()`**

Values *on?* A boolean value.

Description Returns **`true`** if the threads that make up MLWorks are being interleaved via regular pre-emptions.

start *Function*

Summary Switches on pre-emption mode.

Structure **`MLWorks.Threads.Internal.Preemption`**

Syntax **`start () -> ()`**

Arguments **`()`**

Values **`()`**

Description Switches on pre-emption mode.

stop *Function*

Summary Switches off pre-emption mode.

Structure **`MLWorks.Threads.Internal.Preemption`**

Syntax **`stop () -> ()`**

Arguments **`()`**

Values **`()`**

Description Switches off pre-emption mode.

3.5.1 Mutual exclusion primitives

The mutexes implementation can be fount in the **`utils`** subdirectory. The distribution contains the object files **`mutex.mo`** and **`__mutex.mo`**, plus the signature **`MUTEX`**. These objects are not part of the bases, and are therefore not automatically loaded into the image.

Mutexes work with the thread mechanism running in preemption mode.

There are five different demonstration programs in **MLWorks\examples\threads**:

```
dining_philosophers.sml
sleeping_barber.sml
bounded_buffer.sml
cigarette_smokers.sml
readers_writers.sml
```

Documentation of the mutex interface follows.

Mutex *Exception*

Description The exception of mutexes.

mutex *Type*

Description The type of mutexes.

newCountingMutex *Function*

Syntax **newCountingMutex** ***counter*** **->** ***mutex***

Description Returns a new counting mutex with initial value *counter*, an **int**. The *mutex* returned is of type **mutex**.

newBinaryMutex *Function*

Syntax **newBinaryMutex** ***isClaimed*** **->** ***mutex***

Description Returns a new binary mutex with initial value *isClaimed*, a **bool**. The *mutex* returned is of type **mutex**.

test *Function*

Syntax

`test` ***mutex-list*** **`->`** ***bool***

Description

Returns **`true`** if all mutexes in *mutex-list* (type **`mutex list`**) are free at the time of the call, and **`false`** otherwise. It does not block.

testAndClaim *Function*

Syntax

`testAndClaim` ***mutex-list*** **`->`** ***bool***

Description

Like **`test`**, but also claims the mutexes in *mutex-list* if they are all available, returning **`true`**. Returns **`false`** otherwise.

wait *Function*

Syntax

`wait` ***mutex-list*** **`-> ()`**

Description

Blocks the current thread until all mutexes in *mutex-list* (type **`mutex list`**) are simultaneously free. Returns **`unit`**.

Note that a blocked thread will be added to the waiting list of only one of the mutexes.

signal *Function*

Syntax

`signal` ***mutex-list*** **`-> ()`**

Description

Signals that all the mutexes in *mutex-list* (type **`mutex list`**) are free, waking every thread waiting on the mutexes in the list. Returns **`unit`**.

query *Function*

Syntax

`query` ***mutex*** **`->`** ***thread-list***

Description

Returns the list of the threads that are waiting on *mutex*. The *thread-list* is of type **`MLWorks.Threads.Internal.thread_id list`**.

allSleeping *Function*

Syntax

`allSleeping` ***thread-list*** **`-> bool`**

Description

Returns **`true`** if every thread in *thread-list* is sleeping at the time of the call, and **`false`** otherwise. The *thread-list* must be of type **`MLWorks.Threads.Internal.thread_id list`**.

This function can be used for deadlock detection if *thread-list* is the list of every currently running thread.

cleanUp *Function*

`cleanUp () -> ()`

Kills off all threads except those belonging to MLWorks. Takes **`unit`** and returns **`unit`**.

critical *Function*

Syntax

`critical (`***mutex-list*****`,`** ***f*****`)`** ***a*** **`->`** ***value***

Description

Evaluates a function call between a **`wait`** and a **`signal`**, with appropriate behavior on exceptions.

This function first waits for the mutexes in *mutex-list*. Then it applies *f* to *a*. Then it signals that the mutexes in *mutex-list* are free. Finally, it returns the result of the application of *f* to *a*.

The *mutex-list* is of type **`mutex list`**; *f* is a function (type **`'a -> 'b`**) and *a* must be a valid argument to *f*.

await *Function*

Syntax

`await (`*mutex-list*`, `*c*`) -> ()`

Description

Waits until every mutex in *mutex-list* is free and an arbitrary condition is true. The **`await`** function calls c to test for the condition.

If *c*`()` evaluates to **`true`**, **`await`** does *not* release the mutexes. The mutexes are only released if an exception occurs during the evaluation of *c*`()`.

The *mutex-list* is of type **`mutex list`**. The *c* function is of type **`unit -> bool`**. Returns **`unit`**.

4

The MLWorks Interactive Environment Library

4.1 Introduction

The MLWorks interactive environment library is built in to MLWorks, in the permanently available structure **`Shell`**. It provides a number of facilities for customizing the behavior of the MLWorks interactive environment.

The interactive environment library is only available in the interactive context, that is, it is *not* available to the MLWorks batch compiler, and no separate file versions of its units are provided.

The **`Shell`** structure has the following skeletal signature:

```
signature Shell =
 sig
   structure Debug
   structure Dynamic
   structure Editor
   structure Inspector
   structure Options
   structure Path
   structure Profile
   structure Project
   structure Timer
   structure Trace
   val exit: int -> unit
   val saveImage: (string * bool) -> unit
```

```
    val startGUI: unit -> unit
  end
```

The top-level functions **exit**, **saveImage** and **startGUI** relate to the interactive environment.

The **Project** structure provides a programmatic interface to the file-based portion of the MLWorks project system. Much of the functionality provided by these structures is also available in the GUI environment via the project workspace.

The structures **Debug** and **Trace** provide an interface to parts of the MLWorks debugger and tracer tools, while the structures **Profile** and **Time** provide an interface to the profiler. The **Inspector** structure provides an interface to the inspector tool.

The **Options** structure controls the settings of many optional features and parameters of the MLWorks interactive environment, such as the way ML values are printed, and whether compilation should be compatible with features of other ML implementations or of earlier versions of the Definition of Standard ML.

The **Editor** structure provides an interface to the MLWorks custom editor facility.

Finally, the **Dynamic** structure provides an experimental implementation of dynamic types.

4.2 Top-level items: exit, saveImage and startGUI

The functions **exit**, **saveImage** and **startGUI** are defined at the top level of the **Shell** structure.

exit *Function*

Structure	**Shell**	
Syntax	**exit *status* -> ()**	
Arguments	*status*	An integer exit status value.

Values `()`

Description When running MLWorks in TTY mode, **`exit`** kills MLWorks, returning exit status *status*.

When running a GUI listener, **`exit`** simply destroys the listener, without killing MLWorks.

See also the Standard ML Basis function **`OS.Process.exit`**.

saveImage *Function*

Structure **`Shell`**

Syntax **`saveImage (`*filename*`,` *executable?*`) -> ()`**

Arguments

filename	A string naming a file.
executable?	A boolean value.

Values `()`

Description Saves the current session in the image file *filename*.

If *executable?* is **`true`** it saves an executable, otherwise it saves an image file.

If this function is called from the GUI, then when you restart the saved image, the GUI is restarted automatically.

A saved session takes the normal MLWorks command-line arguments.

startGUI *Function*

Structure **`Shell`**

Syntax **`startGUI () -> ()`**

Arguments **`()`**

Values **`()`**

Description If MLWorks was started in TTY mode (that is, **`mlworks`** or **`mlworks-basis`** was started with argument **`-tty`**), this function allows you to start the GUI environment.

If you start the GUI interface with this function, then on exiting the GUI using **File > Exit**, the usual exit dialog also offers an **End X session** button which returns you to TTY mode.

If **`startGUI`** is called with the GUI already running, it prints

```
The MLWorks GUI is already running
```

4.3 Compilation: the Project structure

The **`Project`** structure provides a programmatic interface to the MLWorks project system. The project system is also described in the *MLWorks User Guide*.

4.3.1 Projects and program units

We say that the project compilation system's unit of compilation is a *program unit.* A program unit is a file containing part of an ML program that might be represented in source format or object (compiled) format. If it is a source file it has the name *unit-name*`.sml`, and if it is an object file, it has the name *unit-name*`.mo`.

We call the set of program units that makes up the ML program you are working on a *project*.

4.3.1.1 Specifying pathnames in Shell.Project

Source files, subprojects, and object files all need to have their locations specified by including the directory or folder in which they reside on the relevant path. These pathnames can be specified in either an absolute or relative manner in each function call from **`Shell.Project`**, but you must not mix both notations.

The **`Shell.Project`**functions take filenames in the idiom of the underlying filesystem. For example, **`"~/MLW/basis/_text_io"`** on UNIX and **`"C:\\MLW\\basis\\_text_io"`** for Windows.

Note: The double backslash is necessary on Windows because a single backslash is the Standard ML way to signal an escape sequence. See the Definition.

4.3.2 The Project structure

The **`Project`** structure provides an interface to the project system. **`Project`** has the following skeletal signature:

```
structure Project:
 sig
   eqtype about_details = {description: string, version: string}
   eqtype configuration_details = {library: string list, source:
          string list}
   eqtype location_details = {binariesLoc: string, libraryPath:
          string list,objectsLoc: string}
   eqtype mode_details = {generate_debug_info: bool,
          generate_interceptable_code: bool,
          generate_interruptable_code: bool,
          generate_variable_debug_info: bool,
          location: string,
          mips_r4000: bool, optimize_leaf_fns: bool,
          optimize_self_tail_calls: bool,
          optimize_tail_calls: bool, sparc_v7: bool}

   exception ProjectError of string

   val newProject : unit -> unit
   val openProject : string -> unit
   val saveProject : string -> unit
   val closeProject : unit -> unit
```

```
  val setConfiguration : string -> unit
  val setConfigurationDetails : (string * configuration_details)
      -> unit
  val removeConfiguration : string -> unit
  val showConfigurationDetails : string -> configuration_details
  val showCurrentConfiguration : unit -> string
  val showAllConfigurations : unit -> string list

  val setMode : string -> unit
  val setModeDetails : (string * mode_details) -> unit
  val removeMode : string -> unit
  val showModeDetails : string -> mode_details
  val showCurrentMode : unit -> string
  val showAllModes : unit -> string list

  val setTargets : string list -> unit
  val setTargetDetails : string -> unit
  val removeTarget : string -> unit
  val showCurrentTargets : unit -> string list
  val showAllTargets : unit -> string list

  val showFileName : unit -> string
  val setAboutInfo : about_details -> unit
  val showAboutInfo : unit -> about_details
  val setLocations : location_details -> unit
  val showLocations : unit -> location_details
  val setFiles : string list -> unit
  val showFiles : unit -> string list
  val setSubprojects : string list -> unit
  val showSubprojects : unit -> string list

  val forceLoadAll : unit -> unit
  val forceCompileAll : unit -> unit
  val forceLoad : string -> unit
  val forceCompile : string -> unit
  val readDependencies : string -> unit
  val compile : string -> unit
  val showCompile : string -> unit
  val compileAll : unit -> unit
  val showCompileAll : unit -> unit
  val delete : string -> unit
  val load : string -> unit
  val showLoad : string -> unit
  val loadAll : unit -> unit
  val showLoadAll : unit -> unit
end
```

about_details *Eqtype*

Structure `Shell.Project`

Type `eqtype about_details = {description: string, version: string}`

Description This type is used to store description and version information about a project.

configuration_details *Eqtype*

Structure `Shell.Project`

Type `eqtype configuration_details = {library: string list, source: string list}`

Description This type is used to store configuration details about a project.

location_details *Eqtype*

Structure `Shell.Project`

Type `eqtype location_details = {binariesLoc: string, libraryPath: string list, objectsLoc: string}`

Description This type stores information on the locations of binaries, objects, and the library path.

mode_details *Eqtype*

Structure `Shell.Project`

Type
`eqtype mode_details = {generate_debug_info: bool, generate_interceptable_code: bool, generate_interruptable_code: bool, generate_variable_debug_info: bool, location: string, mips_r4000: bool, optimize_leaf_fns: bool, optimize_self_tail_calls: bool, optimize_tail_calls: bool, sparc_v7: bool}`

Description
This type stores the mode details of a project, such as its optimization settings, the generation of debugging information, and so on.

ProjectError *Exception*

Structure
`Shell.Project`

Type
`exception ProjectError of string`

Description
The exception raised by **`Project`** functions on project errors.

newProject *Function*

Structure
`Shell.Project`

Syntax
`newProject () -> ()`

Arguments
`()`

Values
`()`

Description
Creates a new project.

openProject *Function*

Structure	**`Shell.Project`**	
Syntax	**`openProject`** ***filename*** **`-> ()`**	
Arguments	*filename*	A string.
Values	**`()`**	
Description	Opens the project with the filename given by *filename*.	

saveProject *Function*

Structure	**`Shell.Project`**	
Syntax	**`saveProject`** ***filename*** **`-> ()`**	
Arguments	*filename*	A string.
Values	**`()`**	
Description	Saves the current project in the file specified by *filename*.	

closeProject *Function*

Structure	**`Shell.Project`**
Syntax	**`closeProject () -> ()`**
Arguments	**`()`**
Values	**`()`**

Description — Closes the current project.

setConfiguration *Function*

Structure — **`Shell.Project`**

Syntax — **`setConfiguration`** ***configuration*** **`-> ()`**

Arguments — *configuration* A string.

Values — **`()`**

Description — Sets the current configuration to *configuration*.

See also — **`configuration_details`**

setConfigurationDetails *Function*

Structure — **`Shell.Project`**

Syntax — **`setConfigurationDetails (`*config*`,` *details*`) -> ()`**

Arguments —
config A string.
details A value of type **`configuration_details`**

Values — **`()`**

Description — Sets the details of the configuration specified by *config* to the value given by *details*. If the configuration specified by *config* does not exist then it is created.

removeConfiguration *Function*

Structure	**`Shell.Project`**	
Syntax	**`removeConfiguration`** ***config*** **`-> ()`**	
Arguments	*config*	A string.
Values	**`()`**	
Description	Removes the configuration specified by *config* from the configuration list.	

showConfigurationDetails *Function*

Structure	**`Shell.Project`**	
Syntax	**`showConfigurationDetails`** ***config*** **`->`** ***details***	
Arguments	*config*	A string.
Values	*details*	A value of type **`configuration_details`**.
Description	Shows the details of the configuration specified by *config*.	

showCurrentConfiguration *Function*

Structure	**`Shell.Project`**
Syntax	**`showCurrentConfiguration () ->`** ***config***
Arguments	**`()`**

Values *config* A string.

Description This function returns the name of the current configuration.

showAllConfigurations *Function*

Structure **`Shell.Project`**

Syntax **`showAllConfigurations () ->`** ***configs***

Arguments **`()`**

Values *configs* A list of strings

Description This function returns a list of all defined configurations.

setMode *Function*

Structure **`Shell.Project`**

Syntax **`setMode`** ***mode*** **`-> ()`**

Arguments *mode* A string.

Values **`()`**

Description Sets the mode of the project to *mode*.

setModeDetails *Function*

Structure **`Shell.Project`**

Syntax **`setModeDetails (`*`mode`*`, `*`details`*`) -> ()`**

Arguments *mode* A string.

details A value of type **`mode_details`**.

Values **`()`**

Description Sets the details of the mode specified by *mode* to the values specified by *details*.

removeMode *Function*

Structure **`Shell.Project`**

Syntax **`removeMode `*`mode`*` -> ()`**

Arguments *mode* A string.

Values **`()`**

Description Removed the mode specified by *mode* from the current project.

showModeDetails *Function*

Structure **`Shell.Project`**

Syntax **`showModeDetails `*`mode`*` -> `*`details`***

Arguments *mode* A string.

Values *details* A value of type **`mode_details`**.

Description This function returns the details of the mode specified by *mode*.

showCurrentMode *Function*

Structure **`Shell.Project`**

Syntax **`showCurrentMode () ->`** ***mode***

Arguments **`()`**

Values *mode* A string.

Description This function returns the name of the current mode of the project.

showAllModes *Function*

Structure **`Shell.Project`**

Syntax **`showAllModes () ->`** ***modes***

Arguments **`()`**

Values *modes* A list of strings.

Description This function returns a list of all the names of the modes available in the current project.

setTargets *Function*

Structure **`Shell.Project`**

Syntax **`setTargets`** ***targets*** **`-> ()`**

Arguments *targets* A list of strings.

Values **`()`**

Description — This function sets the list of target sources *targets*.

setTargetDetails *Function*

Structure — **`Shell.Project`**

Syntax — **`setTargetDetails`** ***details*** **`-> ()`**

Arguments — *details* — A string.

Values — **`()`**

Description — This function adds the target specified by *details* to the target list.

removeTarget *Function*

Structure — **`Shell.Project`**

Syntax — **`removeTarget`** ***target*** **`-> ()`**

Arguments — *target* — A string.

Values — **`()`**

Description — This function removes the target source specified by *target* from the list of target sources.

showCurrentTargets *Function*

Structure — **`Shell.Project`**

Syntax — **`showCurrentTargets () ->`** ***targets***

Arguments	**`()`**	
Values	*targets*	A list of strings.
Description	This function returns a list of the names of the currently enabled target sources.	

showAllTargets *Function*

Structure	**`Shell.Project`**	
Syntax	**`showAllTargets () ->`** ***targets***	
Arguments	**`()`**	
Values	*targets*	A list of strings.
Description	This function returns a list of all the target sources in the project, whether enabled or not.	

showFileName *Function*

Structure	**`Shell.Project`**	
Syntax	**`showName () ->`** ***name***	
Arguments	**`()`**	
Values	*name*	A string.
Description	This function returns the filename under which the current project is saved.	

setAboutInfo *Function*

Structure `Shell.Project`

Syntax **`setAboutInfo`** ***details*** **`-> ()`**

Arguments *details* A value of type **`about_details`**.

Values **`()`**

Description This function sets the about details of the current project equal to *details*.

showAboutInfo *Function*

Structure `Shell.Project`

Syntax **`showAboutInfo () ->`** ***details***

Arguments **`()`**

Values *details* A value of type **`about_details`**.

Description This function returns the about details of the current project.

setLocations *Function*

Structure `Shell.Project`

Syntax **`setLocations`** ***details*** **`-> ()`**

Arguments *details* A value of type **`location_details`**.

Values **`()`**

Description — This functions is used to set the location details for the projects, objects and specify the library path for the project.

showLocations *Function*

Structure — **`Shell.Project`**

Syntax — **`showLocations () ->`** ***details***

Arguments — **`()`**

Values — *details* — A value of type **`location_details`**.

Description — This function returns the location details for the current project.

setFiles *Function*

Structure — **`Shell.Project`**

Syntax — **`setFiles`** ***sources*** **`-> ()`**

Arguments — *sources* — A list of strings.

Values — **`()`**

Description — This function sets the list of required files equal to the value of *sources*. Target sources for the project may require files using a statement like:

```
require "foo.sml"
```

The **`setFiles`** function is used to tell the project where to find these required files.

showFiles *Function*

Structure **`Shell.Project`**

Syntax **`showFiles () ->`** ***pathlist***

Arguments **`()`**

Values *pathlist* A list of strings

Description This function returns a list of files required for the project. This list is set using **`setFiles`**.

setSubprojects *Function*

Structure **`Shell.Project`**

Syntax **`setSubprojects`** ***subprojects*** **`-> ()`**

Arguments *subprojects* A list of strings.

Values **`()`**

Description Sets the list of subprojects of the current project equal to *subprojects*.

showSubprojects *Function*

Structure **`Shell.Project`**

Syntax **`showSubprojects () ->`** ***subprojects***

Arguments **`()`**

Values *subprojects* A list of strings.

Description — This function returns a list of the subprojects of the current project.

forceLoadAll *Function*

Structure — **`Shell.Project`**

Syntax — **`forceLoadAll () -> ()`**

Arguments — **`()`**

Values — **`()`**

Description — This function forces the project system to load all compiled object files into the listener.

forceCompileAll *Function*

Structure — **`Shell.Project`**

Syntax — **`forceCompileAll () -> ()`**

Arguments — **`()`**

Values — **`()`**

Description — This function forces the project system to recompile all target sources, whether they have previously been compiled or not.

forceLoad *Function*

Structure — **`Shell.Project`**

Syntax **`forceLoad`** ***object*** **`-> ()`**

Arguments *object* A string.

Values **`()`**

Description This function forces the project system to load the object specified by *object* into the listener.

forceCompile *Function*

Structure **`Shell.Project`**

Syntax **`forceCompile`** ***source*** **`-> ()`**

Arguments *source* A string.

Values **`()`**

Description This function forces the project system to recompile the target source specified by *source*, even if it has been previously compiled.

readDependencies *Function*

Structure **`Shell.Project`**

Syntax **`readDependencies`** ***file*** **`-> unit`**

Arguments *file* A string.

Values **`()`**

Description: This function reads the dependencies of the unit specified by *file*, and sends the output to the console pane of the project workspace.

compile *Function*

Structure: **`Shell.Project`**

Syntax: **`compile`** ***source*** **`-> ()`**

Arguments: *source* — A string.

Values: **`()`**

Description: This function compiles the target source files specified by *source*.

showCompile *Function*

Structure: **`Shell.Project`**

Syntax: **`showCompile`** ***string*** **`-> ()`**

Arguments: *string* — A string

Values: **`()`**

Description: Shows the files that would be compiled if **`compile`** were used with the same *files* argument. Note that the files are not actually compiled.

compileAll *Function*

Structure: **`Shell.Project`**

Syntax **`compileAll () -> ()`**

Arguments **`()`**

Values **`()`**

Description This function compiles the entire project. Note that target source files that have not been altered since the last compilation are not recompiled. Use **`force-Compile`** instead.

showCompileAll *Function*

Structure **`Shell.Project`**

Syntax **`showCompileAll () -> ()`**

Arguments **`()`**

Values **`()`**

Description Shows what the result of executing **`compileAll`** would be without actually compiling or loading any files.

delete *Function*

Structure **`Shell.Project`**

Syntax **`delete`** ***source*** **`-> ()`**

Arguments *source* A string.

Values **`()`**

Description
This function removes an object or target source file from the list of objects and files in the project.

load *Function*

Structure
`Shell.Project`

Syntax
`load` ***object*** **`-> ()`**

Arguments
object A compiled object.

Values
`()`

Description
This function loads the compiled object specified by *object* into the listener.

showLoad *Function*

Structure
`Shell.Project`

Syntax
`showLoad` ***loaded*** **`-> ()`**

Arguments
loaded A string.

Values
`()`

Description
Shows which files would be loaded if the **`load`** function were executed with *loaded* as its argument.

loadAll *Function*

Structure
`Shell.Project`

Syntax
`loadAll () -> ()`

Arguments `()`

Values `()`

Description This function loads all the compiled objects in the project into the listener.

showLoadAll *Function*

Structure **`Shell.Project`**

Syntax **`showLoadAll () -> ()`**

Arguments `()`

Values `()`

Description Shows the effect of executing the **`loadAll`** function, without actually loading any files into the listener.

4.3.3 The Path structure

The **`Path`** structure provides an interface to the batch compilation system's source path mechanism. It has the following signature:

```
structure Path:
 sig
   exception PathError of string
   val sourcePath: unit -> string list
   val setSourcePath: string list -> unit
   val pervasive: unit -> string
   val setPervasive: string -> unit
 end
```

PathError *Exception*

Structure **`Shell.Path`**

Type **`exception PathError of string`**

Description The exception raised by **`Path`** functions on path errors.

sourcePath *Function*

Structure **`Shell.Path`**

Syntax **`sourcePath () ->`** ***path***

Arguments None

Values *path* A list of the directories that make up the source path.

Description Returns a list of the directories that make up the source path. By default, the list is **`["."]`**.

This function raises the exception **`PathError`** if any element of any of the paths does not exist on the filesystem.

The paths are given as standard filesystem path-names rather than in the portable notation.

setSourcePath *Function*

Structure **`Shell.Path`**

Syntax **`setSourcePath`** ***directories*** **`-> ()`**

Arguments *directories* A list of directories to make up the source path.

Values **`()`**

Description	Sets the source path to the list *directories*.

pervasive *Function*

Structure	**`Shell.Path`**	
Syntax	**`pervasive () ->`** ***path***	
Arguments	**`()`**	
Values	*path*	A string.
Description	Returns the directory where the MLWorks pervasive library resides. This path is given as standard filesystem pathname rather than in the portable notation.	

setPervasive *Function*

Structure	**`Shell.Path`**	
Syntax	**`setPervasive`** ***pathstring*** **`-> ()`**	
Arguments	*directory*	A string containing a path.
Values	**`()`**	
Description	Sets the directory where the MLWorks pervasive library resides to *directory*. For internal use only.	

4.4 Debugging interface: the Debug structure

There are two interfaces to the debugger: the GUI version and the TTY version. See the *MLWorks User Guide* for details. The functions in **`Debug`** invoke whichever of the two is appropriate to the context used.

To use the debugger on a function, ensure that the function has been compiled with debugging mode on and, usually, with optimizing mode off.

4.4.1 The Debug structure

```
structure Debug:
  sig
    val clear: ('a -> 'b) -> unit
    val clearAll: unit -> unit
    val info: ('a -> 'b) -> string
    val infoAll: unit -> string list
    val status: ('a -> 'b) -> bool
    val stepThrough: ('a -> 'b) -> 'a -> 'b
  end
```

clear *Function*

Structure **Shell.Debug**

Syntax **clear *function* -> ()**

Arguments *function* A function.

Values **()**

Description Clears the debugging information compiled for a function. For internal use only.

clearAll *Function*

Structure **Shell.Debug**

Syntax **clearAll () -> ()**

Arguments **()**

Values **()**

Description
: Clears all debugging information compiled for all functions. For internal use only.

info *Function*

Structure
: **`Shell.Debug`**

Syntax
: **`info`** ***function*** **`->`** ***string***

Arguments
: *function* A function.

Values
: *string* A string.

Description
: Returns a printable representation of the debugging information compiled for a particular function. For internal use only.

infoAll *Function*

Structure
: **`Shell.Debug`**

Syntax
: **`infoAll () ->`** ***strings***

Arguments
: **`()`**

Values
: *strings* A list of strings.

Description
: Returns a printable representation of debugging information compiled for all functions. For internal use only.

status *Function*

Structure
: **`Shell.Debug`**

Syntax `status` ***function*** `->` ***boolean***

Arguments *function* A compiled function.

Values *boolean* A boolean value.

Description Returns **`true`** if *function* was compiled using debug mode.

stepThrough *Function*

Structure **`Shell.Debug`**

Syntax **`stepThrough`** ***function b*** **`-> ()`**

Arguments *function* A function.

b Arguments to apply to *function*.

Values **`()`**

Description Steps through the evaluation of *function* applied to *b*. Invokes either the GUI debugger or the TTY debugger depending on which version of MLWorks is being used.

4.4.2 Trace

The **`Trace`** structure manipulates the trace and breakpoint states of functions. There are two different interfaces to the trace and breakpoint manager. The simple one consists of all the functions below except **`traceFull`**, and corresponds to the GUI interface.

Note that tracepointing a function overrides any breakpoints it may have, and vice versa.

Breakpoints and tracepoints are set with strings that match functions whose names have that string as a prefix. Therefore, breaking and tracing on a name affects every function whose name has that name as a prefix.

The sophisticated interface is provided by the **`traceFull`** function, and for best results should not be used in conjunction with the simple interface.

```
structure Trace:
  sig
    val breakpoint: string -> unit
    val unbreakpoint: string -> unit
    val unbreakAll: unit -> unit
    val trace: string -> unit
    val untrace: string -> unit
    val untraceAll: unit -> unit
    val traceFull: (string * ('a -> 'b) * ('c -> 'd)) -> unit
  end
```

breakpoint *Function*

Structure **`Shell.Trace`**

Syntax **`breakpoint`** ***name*** **`-> ()`**

Arguments *name* A string.

Values **`()`**

Description Sets a breakpoint on a function called *name*. Breakpoints are also set on all functions with identifiers starting with *name*.

unbreakpoint *Function*

Structure **`Shell.Trace`**

Syntax **`unbreakpoint`** ***name*** **`-> ()`**

Arguments	*name*	A string.
Values	**`()`**	
Description	Removes breakpoint from the function named *name*. Breakpoints are also removed from all functions with identifiers starting with *name*.	

unbreakAll *Function*

Structure	**`Shell.Trace`**
Syntax	**`unbreakAll () -> ()`**
Arguments	**`()`**
Values	**`()`**
Description	Removes breakpoints from all functions.

trace *Function*

Structure	**`Shell.Trace`**	
Syntax	**`trace`** ***`name`*** **`-> ()`**	
Arguments	*name*	A string.
Values	**`()`**	
Description	Traces a function with identifier *name*. Whenever the function is evaluated, its call will be printed.	

untrace *Function*

Structure	**`Shell.Trace`**
Syntax	**`untrace`** ***`name`*** **`-> ()`**
Arguments	*name* A string.
Values	**`()`**
Description	Turns off tracing for the function with identifier *name*. When the function is next called for, it will not be traced.

untraceAll *Function*

Structure	**`Shell.Trace`**
Syntax	**`untraceAll () -> ()`**
Arguments	**`()`**
Value	**`()`**
Description	Turns off tracing for all functions.

traceFull *Function*

Structure	**`Shell.Trace`**
Syntax	**`traceFull (`*`name`*`, `*`f`*`, `*`g`*`) -> ()`**
Arguments	*name* A string containing the name of a function to trace.
	f Conditions on which to trace.

g	Conditions on which to breakpoint.

Values `()`

Description

The function **`traceFull`** implements conditional tracing and breakpointing. For example, assuming you have defined a factorial function **`fact`**, to unconditionally trace the factorial function with no breakpointing type:

```
traceFull ("fact", fn _ => true, fn _ =>
false);
```

To trace the function when the argument is less than 3 type:

```
traceFull ("fact", fn n => n < 3, fn _ =>
false);
```

4.5 Profiling: the Profile structure

The **`Profile`** structure provides a programmatic interface to the MLWorks profiler. See Section 3.4 on page 21 for more details.

```
structure Profile:
  sig
    val profile: ('a -> 'b) -> 'a -> 'b
    val profileFull: MLWorks.Profile.options
                     -> ('a -> 'b) -> 'a -> 'b
    val profileSpace: ('a -> 'b) -> 'a -> 'b
    val profileTime: ('a -> 'b) -> 'a -> 'b
    val profileTool: MLWorks.Profile.profile -> unit
  end
```

profile *Function*

Structure **`Shell.Profile`**

Syntax **`profile`** ***f x*** **`->`** ***values***

Arguments	*f*	An ML function.
	x	Arguments to apply *f* to.
Values	*values*	The values returned by *f x*.

Description

Produces an execution profile of *f x*, invoking the GUI profiler tool on that profile. Space and time profiling are both performed.

To perform the profiling this function calls **`MLWorks.Profile.profile`** with a stack-scan frequency of 10 milliseconds and the following selector function:

```
selector = fn _ => manner
```

Where **`manner`** was constructed by passing the following record to **`MLWorks.Profile.make_manner`**:

```
{ time = true,
  space = true,
  calls = false,
  copies = false,
  depth = 0,
  breakdown = [] }
```

See the **`MLWorks.Profile`** structure for more details of profiling. See the *MLWorks User Guide* for details of the GUI profiler tool.

If you are using MLWorks in TTY mode, this function prints the message:

```
Graphical profiler not available in TTY
Listener
```

profileFull *Function*

Structure **`Shell.Profile`**

Syntax **`profileFull`** ***opts f x***

Arguments	*f*	An ML function.
	x	Arguments to apply *f* to.
	opts	Profiler options.
Values	*values*	The values returned by *f x*.

Description

This function is the same as **`Shell.profile`** *f x*, but allows you to set your own profiler options. This makes it correspond exactly to **`MLWorks.Profile.profile`**, page 28.

profileSpace *Function*

Structure **`Shell.Profile`**

Syntax **`profileSpace`** ***f x*** **`->`** ***values***

Arguments	*f*	An ML function.
	x	Arguments to apply *f* to.
Values	*values*	The values returned by *f x*.

Description

This function is the same as **`profile`** *f x*, except that instead of space and time profiling, only space profiling is performed.

profileTime *Function*

Structure **`Shell.Profile`**

Syntax **`profileTime`** ***f x*** **`->`** ***values***

Arguments	*f*	An ML function.

	x	Arguments to apply *f* to.
Values	*values*	The values returned by *f x*.

Description — This function is the same as **`profile`** *f x*, except that instead of space and time profiling, only time profiling is performed.

profileTool *Function*

Structure	**`Shell.Profile`**	
Syntax	**`profileTool`** ***p*** **`-> ()`**	
Arguments	*p*	An execution profile.
Values	**`()`**	

Description — Invokes the GUI profiler tool on a profile generated by the **`MLWorks.Profile.profile`** function.

4.6 Timing: the Timer structure

The **`Timer`** structure is a convenient programmatic interface to the Basis library timers.

```
structure Timer:
  sig
    val time: ('a ->'b) -> 'a ->
              ({gc: time, sys: time, usr: time} * time)
    val timeIterations: int -> ('a -> 'b) -> 'a ->
              ({gc: time, sys: time, usr: time} * time)
    val printTiming: {function: 'a -> 'b, name: string,
                     outputter: string -> unit} -> 'a -> 'b
  end
```

time *Function*

Structure `Shell.Timer`

Syntax **time *f x* -> *t***

Arguments

f	An ML function.
x	Arguments to apply *f* to.

Values

t	The time elapsed.

Description Returns the time taken to apply *f* to *x*.

timeIterations *Function*

Structure `Shell.Timer`

Syntax **timeIterations *n f x* -> *t***

Arguments

n	The number of times to iterate.
f	An ML function to time.
x	Arguments to apply *f* to.

Values

t	The time elapsed.

Description Returns the time taken to perform *n* applications of *f* to *x*.

printTiming *Function*

Structure `Shell.Timer`

Syntax **printTiming {function=*f*, name=*n*, outputter=*o*} *a* -> *g***

Arguments	*f*	A function.
	n	A string.
	a	An argument for *f*.
	o	A function that prints out a string.
Values	*g*	The return value for *f* applied to *a*.

Description

This function returns the value of *f* applied to *a*, and as a side effect it prints the time taken to evaluate the expression.

Example

Define the following function:

```
fun g x = let
  fun f 0 _ = ()
   | f n m = (f m 0; f (n-1) m)
     in  f x x
  end;
```

Below is an example of applying **`printTiming`** to **`g`** with an argument of 1000:

```
MLWorks>printTiming {function=g,
                     name=”hello”,
                     outputter=print} 1000;
Time for hello : 0.54 (user: 0.49, system:
0.00, gc: 0.00)
val it : unit = ()
MLWorks>
```

4.7 Environment options: the Options structure

The **`Options`** structure provides a large number of flags and other parameters that customize the behavior of the MLWorks interactive environment. It contains the following substructures:

- **`Compiler`**
- **`Debugger`**
- **`Internals`**
- **`Language`**

- **`Mode`**
- **`Preferences`**
- **`ValuePrinter`**

These structures correspond closely with the dialogs in the GUI interface. At top-level in **`Options`**, there is an **`option`** type, a function for setting option values (**`set`**), and a function for returning option values (**`get`**).

4.7.1 Top-level items: ’a option, get and set

The type **`’a option`** and the functions **`get`** and **`set`** are available at top level in **`Options`**.

’a option *Equality type*

Structure	**`Shell.Options`**
Type	**`eqtype ’a option`**
Description	The type of options.

get *Function*

Structure	**`Shell.Options`**	
Syntax	**`get`** ***option*** **`->`** ***value***	
Arguments	*option*	An option.
Values	*value*	The value of *option*.
Description	Gets the value of an option and returns it.	

set *Function*

Structure **`Shell.Options`**

Syntax **`set (`*option*`,` *value*`) -> ()`**

Arguments *option* An option.

value The value to set *option* to.

Values **`()`**

Description Sets the value of an option.

4.7.2 The Compiler structure

```
structure Compiler:
    sig
      val generateDebugInfo: bool option
      val generateTraceProfileCode: bool option
      val generateVariableDebugInfo: bool option
      val interruptTightLoops: bool option
      val optimizeHandlers: bool option
      val optimizeLeaffns: bool option
      val optimizeSelfTailCalls: bool option
      val optimizeTailCalls: bool option
      val printCompilerMessages: bool option
    end
```

generateDebugInfo *Compiler option*

Structure **`Shell.Options.Compiler`**

Type **`val generateDebugInfo: bool option`**

Description If **`true`**, all functions subsequently compiled will be suitable for tracing or call-count profiling.

generateTraceProfileCode *Compiler option*

Structure `Shell.Options.Compiler`

Type `val generateTraceProfileCode: bool option`

Description If **`true`**, tracing and call-counting information is included in all subsequently compiled functions. Those functions will also contain debugging information as generated by **`generateDebugInfo`**.

generateVariableDebugInfo *Compiler option*

Structure `Shell.Options.Compiler`

Type `val generateVariableDebugInfo: bool option`

Description If **`true`**, variable debugging information is generated on compilation if **`generateVariableDebugInfo`** is **`true`**.

interruptTightLoops *Compiler option*

Structure `Shell.Options.Compiler`

Type `val interruptTightLoop: bool option`

Description Although SML does not have a loop construct, functions which are tail recursive are compiled to tight loops in optimizing mode. If **`interruptTightLoops`** is **`true`**, all subsequently compiled tail recursive functions will be interruptible with Ctrl+C or the **interrupt** button on the podium.

optimizeHandlers

Compiler option

Structure `Shell.Options.Compiler`

Type `val optimizeHandlers: bool option`

Description Exception handlers are optimized on subsequent compilations if **`optimizeHandlers`** is **`true`**. The stack may or may not be unwound by exception handlers before executing. It is more efficient if the stack is unwound, but information is lost if the stack browser is entered. If **`optimizeHandlers`** is true, then exception handlers unwind the stack, otherwise they do not and the stack is preserved for debugging purposes.

optimizeLeaffns

Compiler option

Structure `Shell.Options.Compiler`

Type `val optimizeLeaffns: bool option`

Description Leaf functions are optimized on subsequent compilations if **`optimizeLeaffns`** is **`true`**. The effect is that no stack frames are created for the leaf functions.

optimizeSelfTailCalls

Compiler option

Structure `Shell.Options.Compiler`

Type `val optimizeSelfTailCalls: bool option`

Description Self tail calls are optimized on subsequent compilations if **`optimizeSelfTailCalls`** is **`true`**.

optimizeTailCalls *Compiler option*

Structure **Shell.Options.Compiler**

Type **val optimizeTailCalls: bool option**

Description Tail calls are optimized on subsequent compilations if **optimizeTailCalls** is **true**.

printCompilerMessages *Compiler option*

Structure **Shell.Options.Compiler**

Type **val printCompilerMessages: bool option**

Description Compiler messages are printed on compilation when **printCompilerMessages** is **true**.

4.7.3 The Debugger structure

```
structure Debugger:
  sig
    val hideAnonymousFrames: bool option
    val hideCFrames: bool option
    val hideDeliveredFrames: bool option
    val hideDuplicateFrames: bool option
    val hideHandlerFrames: bool option
    val hideSetupFrames: bool option
  end
```

hideAnonymousFrames *Debugger option*

Structure **Shell.Options.Debugger**

Type **val hideAnonymousFrames: bool option**

Description: If **true**, hide anonymous stack frames when in the debugger. An anonymous frame is a frame whose function is an ML anonymous function, that is, one defined with **fn** rather than **fun**.

hideCFrames *Debugger option*

Structure: **Shell.Options.Debugger**

Type: **val hideCFrames: bool option**

Description: If **true**, hide all non-ML stack frames, such as C frames, when in the debugger.

hideDeliveredFrames *Debugger option*

Structure: **Shell.Options.Debugger**

Type: **val hideDeliveredFrames: bool option**

Description: If **true**, hide delivered stack frames when in the debugger. A delivered frame is the frame of a delivered function. Delivered functions have information such as debug information and location information stripped out.

hideDuplicateFrames *Debugger option*

Structure: **Shell.Options.Debugger**

Type: **val hideDuplicateFrames: bool option**

Description: If **true**, hide duplicate stack frames when in the debugger. Setting this option hides extra stack frames introduced by the MLWorks stepping and tracing facility. It is unlikely that you will want to see these extra frames.

hideHandlerFrames *Debugger option*

Structure `Shell.Options.Debugger`

Type `val hideHandlerFrames: bool option`

Description If **`true`**, hide stack frames corresponding to exception handlers.

hideSetupFrames *Debugger option*

Structure `Shell.Options.Debugger`

Type `val hideSetupFrames: bool option`

Description If **`true`**, hide setup stack frames when in the debugger. A setup frame is frame whose function is a setup function, that is, a function that is called when a compiled definition is executed.

4.7.4 The Internals structure

```
structure Internals:
  sig
    val showAbsyn : bool option
    val showLambda: bool option
    val showOptLambda: bool option
    val showEnviron : bool option
    val showMir: bool option
    val showOptMir: bool option
    val showMach: bool option
  end
```

showAbsyn *Internals option*

Structure **Shell.Options.Internals**

Type **val showAbsyn : bool option**

Description If **true**, show abstract syntax representation during compilation. The abstract syntax is the first result of compilation from ML down to machine code.

Example When **showAbsyn** is set to **true**, the following is produced during the evaluation of a simple ML expression:

```
MLWorks> val x = 1;
The abstract syntax

(VALdec
 (
  (
   (x int) <Listener #1>:1,5 to 1,9 1)) () ())
val x : int = 1
MLWorks>
```

showLambda *Internals option*

Structure **Shell.Options.Internals**

Type **val showLambda: bool option**

Description If **true**, show unoptimized lambda code during compilation. Unoptimized lambda code is the result of the second stage in the compilation of ML code.

Example When **showLambda** is set to **true**, the following is produced during the evaluation of a simple ML expression:

```
MLWorks> val x = 1;
The unoptimised lambda code

let
  v14196 = 1
in (v14196, INT 1)
end
val x : int = 1
MLWorks>
```

showOptLambda

Internals option

Structure

Shell.Options.Internals

Type

val showOptLambda: bool option

Description

If **true**, show optimized lambda code during compilation. Optimized lambda code is the result of the third stage in the compilation of ML code. With **showOptLamda** set to **true**, both the unoptimized, and the optimized lambda code are shown.

Example

The following optimized lambda code is produced during the evaluation of a simple ML expression:

```
MLWorks> val x = 1;
The optimized lambda code
let
  v14207 = (1, INT 1)
in v14207
end
val x : int = 1
MLWorks>
```

showEnviron

Internals option

Structure

Shell.Options.Internals

Type

val showEnviron : bool option

Description

If **`true`**, show the lambda environment during compilation. The lambda environment is a mapping of lambda values to lambda variables.

Example

An example of the output produced during the evaluation of a simple ML expression is given below:

```
MLWorks> val x = 1;
The environment
 Ftr: { }
 Top: VE: { x --> FIELD 0/1 }
 SE: { }
val x : int = 1
MLWorks>
```

showMir

Internals option

Structure

`Shell.Options.Internals`

Type

`val showMir: bool option`

Description

If **`true`**, show MIR code during compilation. MIR code is the result of the fourth stage of compilation.

Example

The following is the MIR code resulting from the evaluation of a simple ML expression:

```
MLWorks> val x = 1;
The unoptimised intermediate code
MIR code unit
  References
   Local [Tag 3281, position 0]
   External []
   Interpreter vars []
   Interpreter exns []
   Interpreter strs []
   Interpreter funs []
 Values
 Procedure sets
  Procedure set { 3281 }
   Procedure 3281 <Setup>[<Listener #1>:1,1] (
    No spill size information.
    No previous spill size information.
    No stack allocation information.
    Block 3281
     Procedure entry
     MOVE GC11/closure/cclos GC5/argument/carg
      ; Argument to tuple
      ; Argument to tuple
     ALLOC GC~7545 2
     MOVE GC~7546 1
     ST GC~7546 GC~7545 Any:~1
     MOVE GC~7547 1
     ST GC~7547 GC~7545 Any:3
     MOVE GC5/argument/carg GC~7545
     Procedure exit
val x : int = 1
MLWorks>
```

showOptMir

Internals option

Structure

Shell.Options.Internals

Type

val showOptMir: bool option

Description

If **true**, show optimized MIR code during compilation. Optimized MIR code is the result of the fifth stage of compilation.

Example

The following is the optimized MIR code resulting from the evaluation of a simple ML expression:

```
MLWorks> val x = 1;
The optimised intermediate code
MIR code unit
  References
   Local [Tag 3283, position 0]
   External []
   Interpreter vars []
   Interpreter exns []
   Interpreter strs []
   Interpreter funs []
 Values
 Procedure sets
  Procedure set { 3283 }
   Procedure 3283 <Setup>[<Listener #1>:1,1] (
    Spill areas: GC 0, NON_GC 0, FP 0
    No previous spill size information.
    0 words of stack required.
    Block 3283
     Procedure entry
      ; Argument to tuple
      ; Argument to tuple
     ALLOC GC5/argument/carg 2
     MOVE GC26/g7 1
     ST GC26/g7 GC5/argument/carg Any:~1
     MOVE GC6/i2 1
     ST GC6/i2 GC5/argument/carg Any:3
     MOVE GC5/argument/carg GC5/argument/carg
     Procedure exit
val x : int = 1
MLWorks>
```

showMach

Internals option

Structure

Shell.Options.Internals

Type

val showMach: bool option

Description

If **true**, show object code during compilation. The object code is the final result of the ML compilation procedure. Once generated, the object code is executed.

Example

The following is an example of the output generated during the evaluation of a simple ML expression with **showMach** set to **true**:

```
MLWorks> val x = 1;
The final machine code
[Sparc_Assembly Code] for <Setup>[<Listener
#1>:1,1]
  taddcctv alloc, alloc, #8 ; Attempt to
allocate some heap
  or       arg, limit, #1 ; Tag allocated
pointer
  add      limit, limit, #8 ; Advance
allocation point
  mov      g7, #4
  st       g7, [arg, #~1]
  mov      o2, #4
  retl ; Ordinary return
  st       o2, [arg, #3]
val x : int = 1
MLWorks>
```

4.7.5 The Language structure

```
structure Language:
 sig
  val oldDefinition: bool option
  val abstractions: bool option
  val opOptional: bool option
  val limitedOpen : bool option
  val weakTyvars: bool option
  val fixityInSignatures: bool option
  val fixityInOpen: bool option
  val requireReservedWord: bool option
  val typeDynamic: bool option
end
```

oldDefinition

Compatibility option

Structure

Shell.Options.Language

Type
`val oldDefinition: bool option`

Description
If **`true`**, use the 1990 definition of ML, as in the Definition. When **`oldDefinition`** is **`true`**, new features that are compatible with the old definition are not suppressed, but where there is a conflict between old and new, the old version is implemented.

abstractions

Compatibility option

Structure
`Shell.Options.Language`

Type
`val abstractions: bool option`

Description
If **`true`**, use abstractions. Abstractions are discussed in the *Standard ML Technical Report* by Robert Harper, David MacQueen and Robert Milner, published by the University of Edinburgh. Note that the same functionality is provided in SML '97 by opaque signature matching.

opOptional

Compatibility option

Structure
`Shell.Options.Language`

Type
`val opOptional: bool option`

Description
If **`true`**, set **`op`** as optional when unambiguous. The Definition of Standard ML declares that when an operator identifier is used as something other than an operator — for example, using an operator identifier as a variable identifier as well — it must be preceded by the reserved word **`op`**. However, there are cases in which the usage is unambiguous, and **`op`** is therefore redundant.

Example

The following example demonstrates the use of **opOptional** for an unambiguous datatype:

```
MLWorks> infix &&&;
MLWorks> datatype Foo = &&&;
MLWorks>
```

An error browser containing the message "**Need an op for &&&"** appears. The evaluation fails because **&&&** is not prefixed with an **op**. However, it is clearly an unambiguous case. With **opOptional** set to **true**, MLWorks accepts it:

```
MLWorks> set (opOptional, true);
val it : unit = ()
MLWorks> datatype Foo = &&&;
datatype Foo =
  &&&
val &&& : Foo
MLWorks>
```

Further, supplying an **op** with this option set to **true** will produce a warning, reminding you that it is unnecessary.

limitedOpen

Compatibility option

Structure

Shell.Options.Language

Type

val limitedOpen : bool option

Description

If **true**, set limited **open** in signatures. This option controls the semantics of an **open** *structure* specification in a signature. If the option is **false**, then the values in the structure are added to the returned signature structure, otherwise they are not. This option is not used in SML '97, as the **open** specification has been removed from the language.

weakTyvars

Compatibility option

Structure

Shell.Options.Language

Type

val weakTyvars: bool option

Description

If **true**, set the use of weak type variables. This option is only relevant to the SML '90 mode, as imperative and weak type variables are no longer in the ML language. It is included for compatibility with the Standard ML of New Jersey compiler (NJSML).

NJSML provides weak type variables. These are similar to imperative type variables but include a weakness value indicated by a number at the start of the identifier:

```
MLWorks> val v: '1a -> '1a = id;
```

MLWorks ordinarily treats these as normal type variables, because the Definition of SML does not define them as a separate class:

```
MLWorks> fun id x = x;
val id : 'a -> 'a = fn
MLWorks> val id1 : '1foo -> '1foo = id;
val id1 : 'a -> 'a = fn
MLWorks>
```

With **weakTyvars** set to **true**, however, ML converts them into imperative type variables:

```
MLWorks> fun id x = x;
val id : 'a -> 'a = fn
MLWorks> val id1 : '1foo -> '1foo = id;
val id1 : '_a -> '_a = fn
MLWorks>
```

Note that this does not emulate the full NJSML semantics, but simply allows you to parse New Jersey source using this non-standard notation and provides some similar, standard functionality. Treating NJSML's weak type variables as imperative type variables gives a program using them more chance of being typechecked successfully.

fixityInSignatures

Compatibility option

Structure
`Shell.Options.Language`

Type
`val fixityInSignatures: bool option`

Description
If **`true`**, allow fixity specifications in signatures. This option is included for compatibility with the Standard ML of New Jersey compiler (NJSML).

Example
According to the Definition, the following code is invalid:

```
MLWorks> signature S = sig infix %%% end;
MLWorks>
```

An error browser containing the message "**`Fixity declarations in signatures not valid in this mode.`**" appears.

With **`fixityInSignatures`** set to **`true`**, the signature is accepted:

```
MLWorks> set (fixityInSignatures, true);
val it : unit = ()
MLWorks> signature S = sig infix %%% end;
signature S =
  sig
  end
MLWorks>
```

fixityInOpen

Compatibility option

Structure
`Shell.Options.Language`

Type

```
val fixityInOpen: bool option
```

Description

If **true**, allow inclusion of fixity in **open**. This option considers fixity information given in structures when opened. If this option is not checked when they are defined, structures containing fixity information will be accepted, but the fixity information will not be available when they are opened.

Example

The following structure contains fixity information. In the first case no error occurs:

```
MLWorks> structure S = struct infix $$$ end;
structure S =
  struct
  end
MLWorks> open S;
MLWorks> val $$$ = 3;
val $$$ : int = 3
MLWorks>
```

If the fixity information had been recognized when the structure was opened the final expression,

```
val $$$ =3;
```

would have raised an error, as there was no **op** preceding the **$$$** to show that it was not a reference to the operator with the same name. With **fixityInOpen** set to **true**, the fixity information in the structure is used:

```
MLWorks> set (fixityInOPen, true);
val it : unit = ()
MLWorks> structure S = struct infix $$$ end;
structure S =
  struct
  end
MLWorks> open S;
infix 0 $$$
MLWorks> val $$$ = 3;
MLWorks>
```

An error browser with the message **"Reserved word ’op’ required before infix identifier ’$$$’"** appears.

requireReservedWord *Preferences option*

Structure **Shell.Options.Language**

Type **val requireReservedWord: bool option**

Description If **true**, the Standard ML language that MLWorks recognizes is extended to allow the use of the keyword **require**. This keyword is used to express dependencies in source files.

typeDynamic *Preferences option*

Structure **Shell.Options.Language**

Type **val typeDynamic: bool option**

Description If **true**, the language extension for **dynamic** objects is supported. See “Dynamic types: the Dynamic structure” on page 149.

4.7.6 The Mode structure

The **Mode** structure provides compatibility with the new and old definitions of ML, and provides a concise way of setting global optimization and debugging options.

```
structure Mode:
 sig
  val sml'97: unit -> unit
  val sml'90: unit -> unit
  val compatibility: unit -> unit
  val optimizing: bool option
  val debugging: bool option
 end
```

sml'97 *Function*

Structure	`Shell.Options.Mode`
Syntax	`sml () -> ()`
Arguments	`unit`
Values	`unit`
Description	Sets SML '97 mode preference and turns off compatibility with the Standard ML of New Jersey compiler.

sml'90 *Function*

Structure	`Shell.Options.Mode`
Syntax	`sml'90 () -> ()`
Arguments	`unit`
Values	`unit`
Description	Sets SML '90 mode preference. In this mode, MLWorks accepts the Standard ML language as it was defined in *The Definition of Standard ML* by Rob-

ert Milner, Mads Tofte and Robert Harper, published in 1990 by The MIT Press.

compatibility *Function*

Structure **`Shell.Options.Mode`**

Syntax **`compatibility () -> ()`**

Arguments **`unit`**

Values **`unit`**

Description Sets compatibility with the Standard ML of New Jersey compiler.

optimizing *Mode option*

Structure **`Shell.Options.Mode`**

Type **`val optimizing: bool option`**

Description If **`true`**, all subsequent compilations will be optimized.

debugging *Mode option*

Structure **`Shell.Options.Mode`**

Type **`val debugging: bool option`**

Description If **`true`**, all subsequent compilations will include debugging information.

4.7.7 The Preferences structure

The **`Preferences`** structure controls preferred behaviors in the GUI and TTY environments.

```
structure Preferences:
 sig
  val customEditorName: string option
  val editor: string option
  val externalEditorCommand: string option
  val maximumHistorySize: int option
  val maximumErrors: int option
  val useCompletionMenu: bool option
  val useDebugger: bool option
  val useErrorBrowser: bool option
  val useWindowDebugger: bool option
 end
```

customEditorName *Preferences option*

Structure **`Shell.Options.Preferences`**

Type **`val customEditorName: string option`**

Description Specifies the name of the custom editor that you prefer to use. On Windows the default is **`"Wordpad"`**. The value of this option appears, and can be set, in the **Custom Editor Name** text box on the GUI environment's **Preferences > Editor** dialog.

This option has an effect only if the **`editor`** option is set to **`"Custom"`**.

editor *Preferences option*

Structure **`Shell.Options.Preferences`**

Type **`val editor: string option`**

Description

Specifies the editor-support mode for MLWorks. (See Section 4.8 on page 137 for details of editor-support modes.)

Possible values of **`editor`** are **`"EmacsServer"`**, **`"External"`** and **`"Custom"`**.

If it is **`"EmacsServer"`**, MLWorks uses its Emacs-server editor facility to invoke editors.

If it is **`"External"`**, MLWorks uses the external editor command specified in **`externalEditorCommand`** (which is also the value of the **External Editor** text box on the GUI environment's **Preferences > Editor** dialog) to invoke editors.

If it is **`"Custom"`**, MLWorks uses the custom editor specified in **`customEditorName`** (which is also the value of the **Custom Editor Name** text box on the GUI environment's **Preferences > Editor** dialog) to invoke editors.

This option corresponds to the setting of the radio button panel on the GUI environment's **Preferences > Editor** dialog.

externalEditorCommand *Preferences option*

Structure

`Shell.Options.Preferences`

Type

`val externalEditorCommand: string option`

Description

Specifies the command you prefer to use to invoke a one-shot external editor. This option has an effect only if the **`editor`** option is set to **`"External"`**, meaning that you wish to use a one-shot external editor.

On UNIX, the default is

```
"xterm -name VIsual -e vi +%l %f"
```

On Windows it is

```
"C:\\Program Files\Accessories\Wordpad"
```

This value of this option appears, and can be set, in the **External Editor** text box on the GUI environment's **Preferences > Editor** dialog.

maximumHistorySize

Preferences option

Structure

`Shell.Options.Preferences`

Type

`val maximumHistorySize: int option`

Description

The **`maximumHistorySize`** option is an integer used as the maximum number of history items stored in the listener's **History** menu. The **History** menu contains a list of recent declarations previously accepted by the listener. It is 20 by default.

The value of this option appears, and can be set, in the **Maximum history length** text box on the GUI environment's **Preferences > General** dialog.

The value of this option has no effect if you are not in the GUI environment.

maximumErrors

Preferences option

Structure

`Shell.Options.Preferences`

Type

`val maximumErrors: int option`

Description

The **`maximumErrors`** option is an integer setting the maximum number of errors to be listed in an error browser. It has a default value of 30.

The value of this option appears, and can be set, in the **Maximum number of errors** text box on the GUI environment's **Preferences > General** dialog.

The value of this option has no effect if you are not in the GUI environment.

useCompletionMenu *Preferences option*

Structure `Shell.Options.Preferences`

Type `val useCompletionMenu: bool option`

Description If `true`, MLWorks enables the GUI listener's completion dialog.

The listener's Tab-completion mechanism completes a partially typed identifier when you press Tab. If there is only one possible completion, the listener completes the identifier immediately. If there is more than one possible completion, and `useCompletionMenu` is `true`, a dialog from which you can choose one of the possible completions appears.

If `useCompletionMenu` is `false` when there is more than one possible completion, no dialog appears and the listener makes no change to the partially typed identifier.

This option has no effect if you are not using the GUI environment.

The value of this option corresponds to, and can be set via, the **Use completion menu** option button on the GUI environment's **Preferences > General** dialog.

useDebugger

Preferences option

Structure **`Shell.Options.Preferences`**

Type **`val useDebugger: bool option`**

Description

If you are running the GUI environment and this option is **`true`**, MLWorks enters the debugger when a compilation error occurs, or when a raised exception reaches the top level. The kind of debugger MLWorks enters depends on the settings of **`useErrorBrowser`**, page 129, and **`useWindowDebugger`**, page 130.

If you are running the GUI environment and this option is **`false`**, MLWorks does not enter any debugger but instead prints a simple error message and returns a prompt for more input.

This option is **`true`** by default. It has no effect if you are not in the GUI environment.

The value of this option corresponds to, and can be set via, the **Use debugger** option button on the GUI environment's **Preferences > General** dialog.

useErrorBrowser

Preferences option

Structure **`Shell.Options.Preferences`**

Type **`val useErrorBrowser: bool option`**

Description

If **`true`**, then when running the GUI environment, compilation errors are displayed in the error browser. If **`false`**, compilation errors are printed in the listener instead. The option is **`true`** by default.

The value of this option corresponds to, and can be set via, the **Use error browser** option button on the GUI environment's **Preferences > General** dialog.

This option has no effect if you are not in the GUI environment.

useWindowDebugger *Preferences option*

Structure **`Shell.Options.Preferences`**

Type **`val useWindowDebugger: bool option`**

Description If **`true`**, then when running the GUI environment, the stack browser is invoked when a raised exception reaches the top level. If **`false`**, the system enters the TTY debugger instead. The option is **`true`** by default.

The value of this option corresponds to, and can be set via, the **Always use window debugger** option button on the GUI environment's **Preferences > General** dialog.

This option has no effect if you are not using the GUI environment.

4.7.8 The ValuePrinter structure

The **`ValuePrinter`** structure controls how values are printed in the MLWorks environment. For instance, you can force a listener to elide all lists it prints after the *n*th element. By setting **`maximumSeqSize`** to 3, you can enter

```
MLWorks> ["a", "b", "c", "d"];
```

and the listener will print

```
val it : int list = ["a", "b", "c", ..]
```

The **`ValuePrinter`** structure has the skeletal signature:

```
structure ValuePrinter:
 sig
  val showFnDetails: bool option
  val showExnDetails: bool option
  val floatPrecision: int option
  val maximumSeqSize: int option
  val maximumStringSize: int option
  val maximumDepth: int option
  val maximumRefDepth: int option
  val maximumStrDepth: int option
 end
end
```

The TTY environment (UNIX only) is governed by a single set of **`ValuePrinter`** values, while the GUI environment sophisticates the value-printing feature by giving listeners, compilation managers, and the inspector each their own set. See the *MLWorks User Guide* for details of how this works in the GUI.

showFnDetails *ValuePrinter option*

Structure **`Shell.Options.ValuePrinter`**

Type **`val showFnDetails: bool option`**

Description If **`true`**, the results of each subsequently defined function declaration include the line-and-column location in the listener or source file at which the function was defined. The default setting is **`false`**.

The value of this option corresponds to, and can be set via, the **Show function details** option button in a GUI listener, compilation manager, or inspector tool's **View > Value Printer** dialog.

Example The following extract shows the difference in output between a function defined with **`showFnDetails`** set to **`false`**, and set to **`true`**:

```
MLWorks> set (showFnDetails, false);
val it : unit = ()
MLWorks> val foo = fn x => x + 1;
val foo : int -> int = fn
MLWorks> set (showFnDetails, true);
val it : unit -> ()
MLWorks> val foo = fn x => x + 1;
val foo : int = fn[<anon>[<Listener #1>:1,11 to
1,23]]
MLWorks>
```

showExnDetails *ValuePrinter option*

Structure `Shell.Options.ValuePrinter`

Type `val showExnDetails: bool option`

Description If **`true`**, extra details about exceptions are given in the editor browser.

The value of this option corresponds to, and can be set via, the **Show exception details** option button in a GUI listener, compilation manager, or inspector tool's **View > Value Printer** dialog.

floatPrecision *ValuePrinter option*

Structure `Shell.Options.ValuePrinter`

Type `val floatPrecision: int option`

Description The **`floatPrecision`** option is an integer determining the maximum number of significant digits printed for reals. The default value is 10.

The value of this option appears, and can be set, in the **Precision of reals** text box in a GUI listener, compilation manager, or inspector tool's **View > Value Printer** dialog.

Example

Here is an example of setting **floatPrecision** to 4:

```
MLWorks> 1.23456789;
val it : real = 1.23456789
MLWorks> set (floatPrecision, 4);
val it : unit = ()
MLWorks> 1.23456789;
val it : real = 1.235
MLWorks>
```

maximumSeqSize *ValuePrinter option*

Structure

Shell.Options.ValuePrinter

Type

val maximumSeqSize: int option

Description

The **maximumSeqSize** option is an integer determining the maximum number of elements to print for sequence values. Sequences longer than this are elided. The default value is 10.

The value of this option appears, and can be set, in the **Maximum sequence size** text box in a GUI listener, compilation manager, or inspector tool's **View > Value Printer** dialog.

Example

Here is an example of setting **maximumSeqSize** to 4:

```
MLWorks> val foo = [1, 2, 3, 4, 5, 6];
val foo : int list = [1, 2, 3, 4, 5, 6]
MLWorks> set (maximumSeqSize, 4);
val it : unit = ()
MLWorks> val foo = [1, 2, 3, 4, 5, 6];
val foo : int list = [1, 2, 3, 4, ..]
MLWorks>
```

maximumStringSize

ValuePrinter option

Structure

`Shell.Options.ValuePrinter`

Type

`val maximumStringSize: int option`

Description

The **maximumStringSize** option is an integer determining the maximum number of characters to print for string values. The default value is 255.

The value of this option appears, and can be set, in the **Maximum string size** text box in a GUI listener, compilation manager, or inspector tool's **View > Value Printer** dialog.

Example

Here is an example of the result of setting **maximumStringSize** to 5:

```
MLWorks> set (maximumStringSize, 5);
val it : unit = ()
MLWorks> val bar = "He juggled five clubs at a
time.";
val bar : string = "H\..."
MLWorks>
```

maximumDepth

ValuePrinter option

Structure

`Shell.Options.ValuePrinter`

Type

`val maximumDepth: int option`

Description

The **maximumDepth** option is an integer determining the maximum depth at which to print nested sequences. Elements deeper than this are elided. The default value is 7.

The value of this option appears, and can be set, in the **Maximum depth** text box in a GUI listener, compilation manager, or inspector tool's **View > Value Printer** dialog.

Example

Here is an example of the result of setting **maximumDepth** to 2:

```
MLWorks> val b = [ [["H"]], [["C"]] ];
val b : string list list list = [[["H"]],
[["C"]]]
MLWorks> set (maximumDepth, 2);
val it : unit = ()
MLWorks> val b = [ [["H"]], [["C"]] ];
val b : string list list list [[...], [...]]
MLWorks>
```

maximumRefDepth *ValuePrinter option*

Structure

Shell.Options.ValuePrinter

Type

val maximumRefDepth: int option

Description

The **maximumRefDepth** option is an integer determining the maximum number of self-references a function makes to print. The value printer could be sent into an infinite loop trying to print all the self-references without this upper limit. The default value is 3. The **maximumDepth** will override **maximumRefDepth** if it is set to a smaller value.

The value of this option appears, and can be set, in the **Maximum ref depth** text box in a GUI listener, compilation manager, or inspector tool's **View > Value Printer** dialog.

Example

The following example illustrates the effect of **maximumRefDepth**:

```
MLWorks> ref(ref(ref 3));
val it : int ref ref ref = ref(ref(ref(3)))
MLWorks> set (maximumrefDepth, 1);
val it : unit = ()
MLWorks> ref(ref(ref 3));
val it : int ref ref ref = ref(_)
MLWorks>
```

maximumStrDepth *ValuePrinter option*

Structure `Shell.Options.ValuePrinter`

Type `val maximumStrDepth: int option`

Description The **`maximumStrDepth`** option is an integer determining the maximum depth in a nested structure at which to print ML structure contents. The contents of structures deeper than this are elided. The default value is 2.

The value of this option appears, and can be set, in the **Maximum structure depth** text box in a GUI listener, compilation manager, or inspector tool's **View > Value Printer** dialog.

Example In the following example the value of **`maximumStrDepth`** is the default value of 2. The example structure has depth 3, and therefore the deepest part of it, **`structure U`**, is elided:

```
MLWorks> structure S = struct
                        val x =45;
                        structure T = struct
                          structure U = struct
                           val y = 1; end end
end;
structure S =
  struct
    structure T =
      struct
        structure U = struct ... end
      end
    val x : int = 45
  end
MLWorks>
```

4.8 Editor support: The Editor structure

The MLWorks interface to editor facilities is provided by the **`Editor`** structure, along with the editor items in the structure **`Shell.Options.Preferences`**. MLWorks allows you to edit a named file or to specify that you would like to edit a particular declaration visible in the interactive context, in which case MLWorks searches its compilation records to find the file containing that declaration, then lets you edit it.

MLWorks supports two kinds of editor facility: an Emacs server based on **`emacsclient`**, and the power to invoke any editor by sending to the underlying operating system a command-line call to an editor program. You can define a set of the latter command-line calls and give them names for easy access: they are called *custom editors*.

The command-line strings can use the following codes:

`%f`	A filename.
`%l` and **`%sl`**	Starting line number.
`%c` and **`%sc`**	Starting character number.
`%el`	Ending line number.
`%ec`	Ending character number.

They can use **`%%`** for the literal **`%`**.

For example, on Windows the **Notepad** editor could be defined by:

```
Shell.Editor.Custom.addCommand("My editor",
                    "C:\\Program Files Accessories\\Notepad.exe %f");
```

This command would start **Notepad**. The **%f** code is substituted with a file name, allowing MLWorks to start the editor on a particular file.

Custom editors are similar to external editors because they are defined with a command-line string. However, a custom editor also has:

- a name
- a *connect dialog*

You can use the custom editor's name in the **Preferences > Editor** dialog to specify that you want to use it. The *connect dialog* is not a window-system dialog, but a dialog in the sense of an episode of communication: a dialog consists of a list of commands to be sent to the editor when MLWorks establishes contact with it, and a protocol by which to send the commands. When MLWorks wants to connect to an (existing) remote editor, it first establishes contact and then sends the editor a sequence of commands down the channel that has been established, according to the protocol. These commands typically direct the editor to load a source file at a specific position. Note that connect dialogs are not necessary for all custom editors you define; most editors cannot take commands in this way.

The Emacs server editor option itself uses the connect dialog mechanism to issue the startup commands it needs to make Emacs load in a source file and place the cursor at a specific line. It issues a sequence of Emacs commands including Find File and Goto Line to do so. The example in the entry for **addConnectDialog**, page 140, shows how this can be done.

MLWorks knows a connect dialog style for Windows, based on the DDE (Dynamic Data Exchange) mechanism and called **DDE**.

The **Shell.Editor.Custom.addConnectDialog** function adds a series of dialog commands for a particular custom editor. When MLWorks tries to talk to the selected editor, this sequence of commands (after replacement of any **%** codes) is sent.

The **Editor** structure has the following skeletal signature:

```
structure Editor:
 sig
  structure Custom:
   sig
    val addCommand: (string * string) -> unit
    val addConnectDialog: (string * string * string list) -> unit
    val names: unit -> string list
    val remove: string -> (string * string * string list)
   end
  exception EditError of string
  val editDefinition: ('a -> 'b) -> unit
  val editFile: string -> unit
 end
```

addCommand *Function*

Structure **Shell.Editor.Custom**

Syntax **addCommand (*name*, *command-line*) -> unit**

Arguments

name	A custom editor name.
command-line	A command-line invocation string to associate with custom editor *n*.

Values **()**

Description

Creates a custom editor by associating the editor *name* with a command-line invocation string *command-line*.

The *command-line* string may use the following codes:

%f	A filename.
%l and **%sl**	Starting line number.
%c and **%sc**	Starting character number.
%el	Ending line number.

`%ec`	Ending character number.

Use **`%%`** for the literal **`%`**.

Example

The Windows **`Notepad`** editor could be defined by:

```
Shell.Editor.Custom.addCommand("Notepad",
                               "Notepad %f");
```

addConnectDialog *Function*

Structure

`Shell.Editor.Custom`

Syntax:

`addConnectDialog (`*`name`*`, `*`dialog-name`*`, `*`commands`*`) -> ()`

Arguments

name	A name to be associated with a dialog.
dialog-name	A dialog name.
commands	A list of editing commands.

Values

`()`

Description

Associates the name *name* with the editor connect-dialog *dialog-name* and with a list of editing commands *commands*. The value of *dialog-name* can be either **`"Emacs"`** for Emacs or **`"DDE"`** for DDE.

`"Emacs"`	UNIX only. Used for connecting GNU-Emacs-style editors via a sockets mechanism. Despite being called **`"Emacs"`**, this socket-based dialog provides a general way to talk to editors. However, only a few editors can accept commands in this way.

If the ordinary Emacs server behavior does not quite meet your needs, you could change specify a different dialog string. To use this method you must still set up a copy of Emacs to be a server, using the procedure described in Chapter 1 of the *MLWorks User's Guide*.

"DDE" Windows only. Uses the "execute string" form of Dynamic Data Exchange to allow editors to be controlled remotely.

Example

The connect dialog for an Emacs server could be defined by:

```
let
  val find_file = "(find-file \"%f\")"
  val goto_line = "(goto-line %sl)"
  val fwd_char  = "(forward-char %sc)"
  val highlight = "(mlworks-highlight %sl %sc
%el %ec)"
  val raise_win = "(raise-this-window)"
  val full_dialog  = [find_file, goto_line,
fwd_char,
                    highlight, raise_win]
in
  Shell.Editor.Custom.addConnectDialog("My
Emacs",
                                    "Emacs",

full_dialog)
end
```

The Windows editor PFE could be defined by:

```
let
  val DDE_service = "PFE32"
  val DDE_topic   = "Editor"
  val file_open   = "[FileOpen(\"%f\")]"
  val file_visit  = "[FileVisit(\"%f\")]"
  val goto_line   = "[EditGotoLine(%sl,0)]"
  val fwd_char    = "[CaretRight(%sc,0)]"
  val highlight   =

"[EditGotoLine(%el,1)][CaretRight(%ec,1)]"
  val edit_dialog = [file_open, goto_line,
fwd_char,
                     highlight]
  val DDE_dialog  =

DDE_service :: DDE_topic :: edit_dialog

in

Shell.Editor.Custom.addConnectDialog("PFE32",
                                    "DDE",

DDE_dialog)
end
```

Connect dialog strings can use the following codes:

`%f`	A filename.
`%l` and `%sl`	Starting line number.
`%c` and `%sc`	Starting character number.
`%el`	Ending line number.
`%ec`	Ending character number.

Use `%%` for the literal `%`.

names *Function*

Structure `Shell.Editor.Custom`

Syntax **`names () ->`** ***names***

Arguments **`()`**

Values *names* A list of custom editor names.

Description Returns a list of the names of custom editors currently registered.

remove *Function*

Structure **`Shell.Editor.Custom`**

Syntax **`remove`** ***name*** **`->`** ***command-line dialog-name commands***

Arguments *name* A custom editor name.

Values

command-line The command-line invocation string used to create instances of the custom editor *name*.

dialog-name The name of the connect dialog used with instances of the custom editor *name*.

commands The list of commands passed to instances of the custom editor *name*.

Description Removes the custom editor called *name* from the list of custom editors. Returns the command-line invocation string, the connect dialog name, and the list of editing commands that were associated with that custom editor, or **`("", "", [])`** if there was no custom editor called *name*.

EditError *Exception*

Structure	**`Shell.Editor`**
Type	**`exception EditError of string`**
Description	The exception raised on editor errors.

editDefinition *Function*

Structure	**`Shell.Editor`**	
Syntax	**`editDefinition`** ***f*** **`-> ()`**	
Arguments	*f*	An ML function.
Values	**`()`**	

Description

Invokes your preferred editor (consulting the value of **`Shell.Options.Preferences.editor`** to do so) on the source (**`.sml`**) file containing the declaration of *f*. MLWorks will attempt to place the insertion cursor at the declaration itself.

If *f* was not declared in a file, or if some other file-related error occurs (such as the file no longer existing), **`editDefinition`** raises exception **`EditError`**.

editFile *Function*

Structure	**`Shell.Editor`**	
Syntax	**`editFile`** ***name*** **`-> ()`**	
Arguments	*name*	The name of a source file.

Values	**`()`**
Description	Invokes your preferred (consulting the value of **`Shell.Options.Preferences.editor`** to do so) on the source (**`.sml`**) file *name*. If *name* does not exist, or if some other file-related error occurs, **`editFile`** raises exception **`EditError`**.

4.9 Inspecting values: the Inspector structure

This section examines the **`Shell`** commands for invoking an MLWorks TTY version of the inspector. Note that the TTY inspector can be invoked in the GUI environment's listener, which means that it is available on Windows as well as UNIX. For an explanation of the GUI inspector tool, see section 1.8 of this manual and Chapter 4 of the *MLWorks User Guide*.

The TTY inspector allows you to navigate the value being inspected recursively. At each stage it prints a numbered list of subvalues of the current value. It supports the following commands:

<*field name***>**	Inspect named field of current value
`p`	Inspect parent value of current
`q`	Quit.

The **`Inspector`** structure provides functions for adding and deleting inspector methods. Briefly, an inspector method is a function of type **`t1 -> t2`** that will be applied whenever an object of type **`t1`** is being inspected. On subsequent inspection, values of type **`t1`** are inspected as if they were of type **`t2`**. Note that inspector methods must be compiled in debugging mode.

If a method is added then the inspector assumes that the method handles the printing and recursive navigation of the relevant data structure.

```
structure Inspector:
  sig
    exception InspectError of string
    val addInspectMethod: ('a -> 'b) -> unit
    val deleteAllInspectMethods: unit -> unit
    val deleteInspectMethod: ('a -> 'b) -> unit
    val inspectIt: unit -> unit
  end
```

InspectError *Exception*

Structure **`Shell.Inspector`**

Type **`exception InspectError of string`**

Description The exception raised on inspector errors.

addInspectMethod *Function*

Structure **`Shell.Inspector`**

Syntax **`addInspectMethod`** ***m*** **-> ()**

Arguments *m* An ML function.

Values **`()`**

Description Adds the inspector method *m* to the inspector. An inspector method is a function of type **`t1 -> t2`** that will be applied whenever an object of type **`t1`** is being inspected. On subsequent inspection, values of type **`t1`** are inspected as if they were of type **`t2`**.This function raises exception **`InspectError`** if *m* was not compiled with debugging information.

Example First define the new datatype **`Foo`** as follows:

```
MLWorks> datatype Foo = FOO of int * bool;
datatype Foo =
  FOO of (int * bool)
val FOO : (int * bool) -> Foo
MLWorks> FOO (10, false);
val it : Foo = FOO (10, false)
MLWorks>
```

Using **inspectIt** returns the following result:

```
MLWorks> inspectIt();
Entering TTY inspector - enter ? for help
Value: FOO (10, false)
Type: Foo
1: 10
2: false
Inspector> q
val it : unit = ()
MLWorks>
```

Compare this output with the following output, resulting from inspecting **Foo** after adding a new inspector method using **addInspectMethod**:

```
MLWorks> fun inspectFoo (FOO (n,b)) = if b then
{value = 0} else {value=n};
val inspectFoo : Foo -> {value: int} = fn
MLWorks> addInspectMethod (inspectFoo);
val it : unit = ()
MLWorks> FOO (10, false);
val it : Foo = FOO (10, false)
MLWorks> inspectIt();
Entering TTY inspector - enter ? for help
Value: FOO (10, false)
Type: Foo
value: 10
Inspector> q
val it : unit = ()
MLWorks>
```

deleteInspectMethod *Function*

Structure **Shell.Inspector**

Syntax	**`deleteInspectMethod`** ***m*** **`-> ()`**
Arguments	*m* — An ML function.
Values	**`()`**
Description	Deletes the inspector method *m* from the inspector.

deleteAllInspectMethods *Function*

Structure	**`Shell.Inspector`**
Syntax	**`deleteAllInspectMethods () -> ()`**
Arguments	**`()`**
Values	**`()`**
Description	Deletes all inspector methods.

inspectIt *Function*

Structure	**`Shell.Inspector`**
Syntax	**`inspectIt: () -> ()`**
Arguments	**`()`**
Values	**`()`**
Description	Invokes the TTY inspector tool on the value of **`it`**. If there is no method for displaying this type, then the exception **`LookupValId`** is raised.

4.10 Dynamic types: the Dynamic structure

An experimental implementation of dynamic types.

```
structure Dynamic:
  sig
    exception Coerce of (type_rep * type_rep)
    exception EvalError of string
    type dynamic
    type type_rep
    val eval: string -> dynamic
    val getType: dynamic -> type_rep
    val printType: type_rep -> string
    val printValue: dynamic -> string
    val inspect: dynamic -> unit
  end
```

EvalError *Exception*

Structure `Shell.Dynamic`

Type `exception EvalError of string`

Description The exception raised on dynamic errors.

dynamic *Type*

Structure `Shell.Dynamic`

Type `type dynamic`

Description The dynamic type.

type_rep *Type*

Structure `Shell.Dynamic`

Type **`type type_rep`**

Description Representation of types.

eval *Function*

Structure **`Shell.Dynamic`**

Syntax **`eval`** ***expr*** **`->`** ***dynamic***

Argument *expr* A string of an ML expression to dynamically evaluate.

Values *dynamic* The dynamic evaluation of *expr*.

Description Given a string ML expression, evaluates it dynamically. If there is a type error, it raises **`EvalError`**.

getType *Function*

Structure **`Shell.Dynamic`**

Syntax **`getType`** ***dynamic*** **`->`** ***type***

Arguments *dynamic* A dynamic value whose type is to be determined.

Values *type* The type of the dynamic value *dynamic*.

Description Given a dynamic value, retrieves its type.

printType *Function*

Structure **`Shell.Dynamic`**

Syntax **`printType`** ***type*** **`->`** ***string***

Arguments *type* A type representation to convert.

Values *string* The returned string form of *type*.

Description Convert type representation into string form.

printValue *Function*

Structure **`Shell.Dynamic`**

Syntax **`printValue`** ***dynamic*** **`->`** ***string***

Arguments *dynamic* A dynamic value to be converted to string form.

Values *string* The returned string form of *dynamic*.

Description Convert the value of a dynamic type into a string.

inspect *Function*

Structure **`Shell.Dynamic`**

Syntax **`inspect`** ***dynamic*** **`-> ()`**

Arguments *dynamic* A dynamic type on which a TTY inspector is to be invoked.

Values	**()**
Description	Invoke the TTY inspector on the value of dynamic type.

5

The MLWorks Motif Interface Library

5.1 Introduction

The MLWorks Motif interface library is provided in the **Xm** structure. The **Xm** structure is not built in to the MLWorks interactive environment, but is distributed as a set of source and separately compiled files with UNIX and Linux versions of MLWorks. See the installation notes for your version of MLWorks for details of their location.

The Motif interface library provides an ML interface to the X Window System with Motif, including relevant parts of the X Toolkit, and to the graphics functionality of Xlib. For the most part, functions here are directly equivalent to the corresponding C functions. Please refer to the Motif documentation for function descriptions. We have taken advantage of the ML type system to make the interface more secure than the C interface.

To use this library in your applications, add

```
require "$.motif.__xm".
```

to the start of the relevant source files. To load the library into the MLWorks interactive environment, set the appropriate library path using the project workspace or use the functions in **Shell.Project**.

5.2 Type conventions

Differences between the ML and C type systems mean that we have had to adopt certain translation conventions in our implementation. This section describes those conventions.

Where a C function "returns" information by assigning values to a pointer argument, the corresponding ML function returns the information as the result value.

Although the result of a C function indicates success or failure, the corresponding ML function does not. Instead, it indicates failure by raising an exception. Functions with return type `unit` do not return a useful value.

Where a C function takes a variable number of arguments, the corresponding ML function takes a list. Where one of a set of predefined options needs to be passed to a function, these options are implemented as datatypes. Resources are also implemented as datatypes.

If a C function takes a disjunction of flags combined in a single word, the corresponding ML function takes a list of values. The possible values are specified by a datatype.

5.3 Naming conventions

The identifiers in this library follow the conventions used in the Standard ML Basis Library.

No distinction is made between functions from Xlib, Xt, and Motif. Instead, a single interface to the relevant functionality is provided. This document gives the corresponding C function for each ML function.

Constants represented by constructors are given the same name as the constants in the C programming interface.

Most operations are grouped into substructures. Each substructure contains those functions that operate on a particular type of object or class of widget.

5.4 Abstract types

The library uses the following abstract types:

Table 5.1 ML equality types and their C equivalents

ML equality type	Equivalent C type
display	**Display***
screen	**Screen***
widget	**Widget**
gc	**GC**
visual	**Visual***
pixel	**Pixel**
font	**Font**
font_struct	**XFontStruct***
font_list	**XmFontList**
compound_string	**XmString**
translations	**XtTranslations**
atom	**Atom**
drawable	**Drawable**
colormap	**Colormap**

5.5 Exceptions

The **XSystemError** exception is used when a C function which is called by one of the functions below fails. It is typically used in functions where the C version of the function returns a result status and the ML implementation of the function returns **unit** or a data value.

```
exception XSystemError of string
```

The **ArgumentType** exception is raised when a function is passed a resource name with an inappropriate resource value. See Section 5.6 for more details.

```
exception ArgumentType of argument_name * argument_value
```

The **NotInitialized** exception is raised if an event handling function is called before the X Window System has been initialized. The **SubLoopTerminated** exception is raised when **doInput** is called when the application has been exited (see Section 5.9).

```
exception NotInitialized
```

```
exception SubLoopTerminated
```

5.6 Resources

Resources are represented by datatypes. A list of resources is a list of pairs of the type:

```
(argument_name * argument_value)
```

argument_name *Datatype*

The **argument_name** type is an enumerated datatype of supported resources. The declaration for **argument_name** is in **motif/xm.ML**.

argument_value *Datatype*

The **argument_value** datatype has the following declaration:

```
datatype argument_value =
     INT of int |
     STRING of string |
     BOOL of bool |
     COMPOUND_STRING of compound_string |
     EDIT_MODE_VALUE of edit_mode |
     SELECTION_POLICY_VALUE of selection_policy |
     WINDOW of drawable |
     PIXMAP of drawable |
     LABEL_TYPE_VALUE of label_type |
     FONT_VALUE of font |
     FONT_LIST_VALUE of font_list |
     PIXEL of pixel |
     ROW_COLUMN_TYPE_VALUE of row_column_type |
     WIDGET of widget |
     ORIENTATION_VALUE of orientation |
     PACKING_VALUE of packing_type |
     DIALOG_TYPE_VALUE of dialog_type |
     SCROLLING_POLICY_VALUE of scrolling_policy |
     SCROLLBAR_DISPLAY_POLICY_VALUE of scrollbar_display_policy |
     ATTACHMENT of attachment |
     DELETE_RESPONSE_VALUE of delete_response
```

The arguments of these constructors are either built-in ML types, abstract types defined in Section 5.4, or enumerated types of the values that can be legally associated with that resource (see the **XM** signature declaration for details).

When an argument list is passed to a function, the **Xm** structure checks that the **(argument name * argument value)** pairs are legal combinations. It does not check whether the **argument_name** is legal for the particular function.

Here is an example:

```
val applicationShell =
  Xm.initialize
  (name, title,
   [(Xm.TITLE, Xm.STRING title),
    (Xm.ICON_NAME, Xm.STRING title)])
```

5.7 Initialization

Xm must be initialized using the **initialize** function. This may be called both interactively and from stand-alone applications. It handles the house-keeping of initializing the X Toolkit, opening the display, and setting the error handlers. It then returns an application shell widget.

The function **`initialize`** takes three arguments: the name of the application, the name of the application class, and a list of resources. These are passed to **`XtOpenDisplay`** if it is called. The first two are also passed to **`XtAppCreateShell`**.

The **`deleteResponse`** resource of the application shell widget is always set to **`DO_NOTHING`** by **`initialize`**.

When running an application from the MLWorks interactive environment, the event dispatch loop in the environment itself will dispatch all events to the application as well. Stand-alone applications must run **`Xm.mainLoop`** to start an event dispatch loop of their own. Applications started from the MLWorks interactive environment must not run this function.

Event-handling subloops may be created using the **`Xm.doInput`** function. This dispatches a single event. It returns **`false`** if the application shell has been destroyed. Otherwise it returns **`true`**.

5.8 Callbacks

Add callbacks with the function **`Xm.Callback.add`**. This takes three arguments: the widget, the name of the callback, and the callback function itself.

The name of the callback is given by a constructor of the type:

```
Xm.Callback.name
```

The callback function argument has the following type:

```
Xm.Callback.callback_data -> unit
```

The **`Xm.Callback.callback_data`** type is an abstract type. The conversion functions in the **`Xm.Callback`** structure convert values of the abstract type to concrete data. The system does not check that the user has called the appropriate conversion function for the particular callback.

There is an extra complication with **`ModifyVerify`** callbacks. If the program needs to prevent the modification from occurring, it must set a boolean value in the C structure. The **`Xm`** structure provides this functionality using the **`Xm.Boolean`** structure. This provides an abstract type that corresponds to the location in the C structure, and a function that can set an occurrence of the abstract type to either **true** or **false**.

Here is an example:

```
val text = Xm.Widget.create ...

fun modifyVerify callback_data =
  let
    val (_,_,doit,_,_,start_pos,end_pos,str) =
      Xm.Callback.convertTextVerify data
  in
    if ... then
      Xm.Boolean.set (doit, true)
    else
      (Xm.Display.bell (Xm.Widget.display text, 0)
       Xm.Boolean.set (doit, false))
  end

Xm.Callback.add (text, Xm.Callback.MODIFY_VERIFY, modifyVerify)
```

5.9 Events

Events are implemented similarly to callbacks. To add an event handler to a widget, use the function **Xm.Event.addHandler**. This corresponds to the C function **XtAddEventHandler**. The event mask is specified by a list of constructors of the **Xm.Event.make** type. The handler function itself takes an argument of the abstract type **Xm.Event.event_data**, which can be converted to concrete data using the **Xm.Event.convertEvent** function.

The **Xm.Event.event** datatype defines the concrete data resulting from converting the abstract type. Particular classes of events are defined by the **expose_event**, **motion_event**, **button_event**, and **key_event** types.

The values of the button and state fields of the **event** datatypes are not automatically converted to ML values, because they aren't always needed. When these values are needed, they must be explicitly converted using the **convertState** and **convertButton** functions.

5.10 Identifier reference

Most functions are grouped into substructures. Each substructure defines operations on a particular type or a particular sort of widget.

The following tables contain mappings from MLWorks Motif interface library identifiers to their corresponding X Window System and Motif identifiers

Table 5.2 Top-level Xm structure identifiers

ML identifier	C identifier
isMotifWMRunning	**XmIsMotifWMRunning**
updateDisplay	**XmUpdateDisplay**
sync	**XSync**
synchronize	**XSynchronize**

Table 5.3 **Display** structure identifiers

ML identifier	C identifier
bell	**Xbell**
queryPointer	**XQueryPointer**

Table 5.4 **Pixel** structure identifiers

ML identifier	C identifier
screenBlack	**BlackPixelOfScreen**
screenWhite	**WhitePixelOfScreen**

Table 5.5 **`Font`** structure identifiers

ML identifier	C identifier
`load`	**`XLoadFont`**
`free`	**`XFreeFont`**
`query`	**`XQueryFont`**
`textExtents`	**`XTextExtents`**

Table 5.6 **`Widget`** structure identifiers

ML identifier	C identifier
`create`	**`XtCreateWidget`**
`createMenuBar`	**`XmCreateMenuBar`**
`createPopupShell`	**`XtCreatePopupShell`**
`createPulldownMenu`	**`XmCreatePulldownMenu`**
`createScrolledText`	**`XmCreateScrolledText`**
`createManaged`	**`Xm.Widget.create`** followed by **`Xm.Widget.manage`**
`destroy`	**`XtDestroyWidget`**
`manage`	**`XtManageChild`**
`map`	**`XtMapWidget`**
`unmanageChild`	**`XtUnmanageChild`**
`unmap`	**`XtUnMapWidget`**
`realize`	**`XtRealizeWidget`**

Table 5.6 **`Widget`** structure identifiers

ML identifier	C identifier
`unrealize`	**`XtUnrealizeWidget`**
`isRealized`	**`XtIsRealized`**
`popup`	**`XtPopup`**
`valuesSet`	**`XtSetValues`**
`valuesGet`	**`XtGetValues`**
`display`	**`XtDisplay`**
`window`	**`XtWindow`**
`screen`	**`XtScreen`**
`parent`	**`XtParent`**
`name`	**`XtName`**
`processTraversal`	**`XmProcessTraversal`**
`toFront`	Brings widget to the front using **`XReconfigureWMWindow`**.

Table 5.7 **`Atom`** structure identifiers

ML identifier	C identifier
`intern`	**`XmInternAtom`**
`getname`	**`XmGetAtomName`**

Table 5.8 **`Pixmap`** structure identifiers

ML identifier	C identifier
`create`	**`XCreatePixmap`**
`free`	**`XFreePixmap`**
`get`	**`XmGetPixmap`**
`destroy`	**`XmDestroyPixmap`**

Table 5.9 **`CascadeButton`** structure identifier

ML identifier	C identifier
`highlight`	**`XmCascadeButtonHighlight`**

Table 5.10 **`CascadeButtonGadget`** structure identifier

ML identifier	C identifier
`highlight`	**`XmCascadeButtonGadgetHighlight`**

Table 5.11 **`CompoundString`** structure identifiers

ML identifier	C identifier
`baseline`	**`XmStringBaseline`**
`byteCompare`	**`XmStringByteCompare`**

Table 5.11 **`CompoundString`** structure identifiers

ML identifier	C identifier
`compare`	**`XmStringCompare`**
`concat`	**`XmStringConcat`**
`copy`	**`XmStringCopy`**
`create`	**`XmStringCreate`**
`createLtoR`	**`XmStringCreateLtoR`**
`createSimple`	**`XmStringCreateSimple`**
`directionCreate`	**`XmStringDirectionCreate`**
`empty`	**`XmStringEmpty`**
`extent`	**`XmStringExtent`**
`free`	**`XmStringFree`**
`hasSubstring`	**`XmStringHasSubstring`**
`height`	**`XmStringHeight`**
`length`	**`XmStringLength`**
`lineCount`	**`XmStringLineCount`**
`nConcat`	**`XmStringNConcat`**
`nCopy`	**`XmStringNCopy`**
`width`	**`XmStringWidth`**
`convertStringText`	no corresponding identifier

The function **`convertStringText`** is a convenience function that extracts a simple string from a compound string.

Table 5.12 **`FileSelectionBox`** structure identifiers

ML identifier	C identifier
`getChild`	**`XmFileSelectionBoxGetChild`**
`doSearch`	**`XmFileSelectionDoSearch`**

Table 5.13 **`FontList`** structure identifiers

ML identifier	C identifier
`add`	**`XmFontListAdd`**
`copy`	**`XmFontListCopy`**
`create`	**`XmFontListCreate`**
`free`	**`XmFontListFree`**

Table 5.14 **`List`** structure identifiers

ML identifiers	C identifiers
`addItem`	**`XmListAddItem`**
`addItemUnselected`	**`XmListAddItemUnselected`**
`addItems`	**`XmListAddItems`**
`deleteAllItems`	**`XmListDeleteAllItems`**
`deleteItem`	**`XmListDeleteItem`**
`deleteItems`	**`XmListDeleteItems`**
`deleteItemsPos`	**`XmListDeleteItemsPos`**

Table 5.14 **`List`** structure identifiers

ML identifiers	C identifiers
`deletePos`	**`XmListDeletePos`**
`getSelectedPos`	**`XmListGetSelectedPos`**
`setBottomPos`	**`XmListSetBottomPos`**
`selectPos`	**`XmListSelectPos`**
`setPos`	**`XmListSetPos`**

Table 5.15 **`MessageBox`** structure identifier

ML identifier	C identifier
`getChild`	**`XmMessageBoxGetChild`**

Table 5.16 **`Scale`** structure identifiers

ML identifier	C identifier
`getValue`	**`XmScaleGetValue`**
`setValue`	**`XmScaleSetValue`**

Table 5.17 **`ScrollBar`** structure identifiers

ML identifier	C identifier
`getValues`	**`XmScrollBarGetValues`**
`setValues`	**`XmScrollBarSetValues`**

Table 5.18 **`ScrolledWindow`** structure identifier

ML identifier	C identifier
`setAreas`	**`XmScrolledWindowSetAreas`**

Table 5.19 **`SelectionBox`** structure identifier

ML identifier	C identifier
`getChild`	**`XmSelectionBoxGetChild`**

Table 5.20 **`TabGroup`** structure identifiers

ML identifier	C identifier
`add`	**`XmAddTabGroup`**
`remove`	**`XmRemoveTabGroup`**

Table 5.21 **`Text`** structure identifies

ML identifier	C identifier
`clearSelection`	**`XmTextClearSelection`**
`copy`	**`XmTextCopy`**
`cut`	**`XmTextCut`**
`getBaseline`	**`XmTextGetBaseline`**

Table 5.21 **`Text`** structure identifies

ML identifier	C identifier
`getEditable`	**`XmTextGetEditable`**
`getMaxLength`	**`XmTextGetMaxLength`**
`getInsertionPosition`	**`XmTextGetInsertionPosition`**
`getLastPosition`	**`XmTextGetLastPosition`**
`getSelection`	**`XmTextGetSelection`**
`getSelectionPosition`	**`XmTextGetSelectionPosition`**
`getString`	**`XmTextGetString`**
`getTopCharacter`	**`XmTextGetTopCharacter`**
`insert`	**`XmTextInsert`**
`paste`	**`XmTextPaste`**
`posToXY`	**`XmTextPosToXY`** Failure returns (-1, -1).
`remove`	**`XmTextRemove`** Ignores return value.
`replace`	**`XmTextReplace`**
`scroll`	**`XmTextScroll`**
`setAddMode`	**`XmTextSetAddMode`**
`setEditable`	**`XmTextSetEditable`**
`setHighlight`	**`XmTextSetHighlight`**
`setInsertionPosition`	**`XmTextSetInsertionPosition`**
`setMaxLength`	**`XmTextSetMaxLength`**
`setSelection`	**`XmTextSetSelection`**

Table 5.21 **Text** structure identifies

ML identifier	C identifier
setString	**XmTextSetString**
setTopCharacter	**XmTextSetTopCharacter**
showPosition	**XmTextShowPosition**
xyToPos	**XmTextXyToPos**

Table 5.22 **ToggleButton** structure identifiers

ML identifier	C identifier
getState	**XmToggleButtonGetState**
setState	**XmToggleButtonSetState**

Table 5.23 **ToggleButtonGadget** structure identifiers

ML identifier	C identifier
getState	**XmToggleButtonGadgetGetState**
setState	**XmToggleButtonGadgetSetState**

Table 5.24 **Translation** structure identifiers

ML identifier	C identifier
parseTable	**XtParseTranslationTable**
override	**XtOverrideTranslations**

Table 5.24 **`Translation`** structure identifiers

ML identifier	C identifier
`augment`	**`XtAugmentTranslations`**
`uninstall`	**`XtUninstallTranslations`**

The following structures provide Xlib graphics capabilities:

Table 5.25 **`GC`** structure identifiers

ML identifiers	C identifiers
`create`	**`XCreateGC`**
`change`	**`XChangeGC`**
`copy`	**`XCopyGC`**
`free`	**`XFreeGC`**
`setClipRectangles`	**`XSetClipRectangles`**
`getValues`	**`XGetGCValues`**

The **`gc_value`** datatype, and the types on which it depends, enumerate the possible values that may be set in a **`GC`**. The **`request`** datatype enumerates the same values for the **`getValues`** function.

Table 5.26 **`Draw`** structure interfaces

ML interface	C interface
`string`	**`XDrawString`**
`imageString`	**`XDrawImageString`**
`line`	**`XDrawLine`**

Table 5.26 **`Draw`** structure interfaces

ML interface	C interface
`lines`	`XDrawLines`
`segments`	`XDrawSegments`
`fillPolygon`	`XFillPolygon`
`point`	`XDrawPoint`
`points`	`XDrawPoints`
`rectangle`	`XDrawRectangle`
`fillRectangle`	`XFillRectangle`
`rectangles`	`XDrawRectangles`
`fillRectangles`	`XFillRectangles`
`arc`	`XDrawArc`
`fillArc`	`XFillArc`
`arcs`	`XDrawArcs`
`fillArcs`	`XFillArcs`
`clearArea`	`XClearArea`
`copyArea`	`XCopyArea`
`copyPlane`	`XCopyPlane`

Table 5.27 **`Colormap`** structure identifiers

ML identifier	C identifier
`default`	`DefaultColormapOfScreen`
`allocColor`	`XAllocColor`

Table 5.27 **`Colormap`** structure identifiers

ML identifier	C identifier
`allocNamedColor`	**`XAllocNamedColor`**
`allocColorCells`	**`XAllocColorCells`**
`freeColors`	**`XFreeColors`**
`storeColor`	**`XStoreColor`**
`storeNamedColor`	**`XStoreNamedColor`**

6

The MLWorks Windows Interface Library

6.1 Introduction

The MLWorks Windows interface library is provided in the structure **`WindowsGui`**. The **`WindowsGui`** structure is not built in to the MLWorks interactive environment, but is distributed as a set of source and separately compiled files with Windows versions of MLWorks. See the installation notes for your version of MLWorks for details of their location.

The Windows interface library provides an ML interface to a selection of Windows SDK functions concerned with window-programming tasks, to allow you to write windowing applications in MLWorks. The library is essentially a mapping of ML identifiers to C identifiers from the Windows SDK. This chapter therefore does not describe the semantics of every ML identifier in the library. For a description of the semantics of a particular ML identifier, refer to the entry in the Windows SDK documentation for the corresponding C identifier.

The ML functions and types that have been defined can mostly be used in the same way as the C functions and types to which they correspond. However, because the ML and C type systems differ, there are exceptions to this rule. For instance, some of the ML functions' parameters and return values differ from those of their C counterparts. Where differences exist between the ML interface and the original C interface, they are documented.

To use the library in your applications, add

```
require "$.mswindows.__windowsgui"
```

to the start of the relevant source files. To load the library into the MLWorks interactive environment, use the **Set Library Path** option in the project workspace, or use the relevant functions in **Shell.Project**.

6.2 Type conventions

Differences between the ML and C type systems mean that we have had to adopt certain translation conventions in our implementation. This section describes those conventions.

Where a C function "returns" information by assigning values to a pointer argument, the corresponding ML function returns the information as the result value.

Where the result of a C function indicates success or failure, the corresponding ML function does not. Instead, the ML function indicates failure by raising an exception. Functions with return type **unit** do not return a useful value.

Where a C function takes a variable number of arguments, the corresponding ML function takes a list. Lists are also used to pass flags, where the corresponding C function combines the flags with the logical OR function.

Values defined by **#define**, such as messages, are represented by ML datatypes.

Where an ML function differs from a C function, in terms of parameters and return values, is marked accordingly.

6.3 Naming conventions

The identifiers in this library follow the conventions used in the Standard ML Basis library.

ML function names are without initial capitalization, but are otherwise the same as the names of the Windows functions they mirror.

Where one of a set of predefined options needs to be passed to a function, these options are implemented as datatypes. These options can also be passed as a disjunction as the Windows function permits — see the SDK documentation. The options shown below are the only ones supported. All options are given the same names as in the Windows SDK.

6.4 Exceptions

WindowSystemError *Exception*

```
exception WindowSystemError of string
```

This exception is used when a C function called from ML fails. It is typically used in functions where the C version of the function returns a boolean value indicating success or failure and the ML implementation of the function returns **`unit`**.

6.5 Data types

The following table contains a list of the relevant datatypes and their corresponding Windows types. All C types are of type **`word`** unless indicated otherwise under notes.

Table 6.1 ML types and their corresponding C types

ML type	C type	Notes
`hwind`	**`HWND`**	Window handle.
`hmenu`	**`HMENU`**	Menu handle.
`word`	**`LONG`**	See “word” on page 176
`hdc`	**`HDC`**	Device context handle.
`accelerator_table`	**`ACCELERATOR_TABLE`**	Table handle.
`wparam`	**`WPARAM`**	

Table 6.1 ML types and their corresponding C types

ML type	C type	Notes
lparam	**LPARAM**	
rect	**RECT**	See “rect” on page 176.
point	**POINT**	See “point” on page 176.
color	**COLOR**	

word *Datatype*

Description

The **word** type is **Word32.word** from the Standard ML Basis library. The Windows **LONG** type is usually implemented as a **word** type in ML.

rect *Datatype*

Type

```
datatype rect = RECT of
        {left:int, top:int, right:int, bottom:
int}
```

point *Datatype*

Type

```
datatype point = POINT of {x:int, y:int}
```

message *Datatype*

Type

```
datatype message =
      BM_GETCHECK |
      BM_GETSTATE |
      BM_SETCHECK |
      …
```

Description

Messages are represented by values of the **message** datatype. Other groups of options are defined by similar datatypes.

window_style *Datatype*

Type

```
datatype window_style =
      BS_3STATE |
      BS_AUTO3STATE |
      BS_AUTOCHECKBOX |
      …
```

sw_arg *Datatype*

Type

```
datatype sw_arg =
      SW_HIDE |
      SW_MAXIMIZE |
      SW_MINIMIZE |
      …
```

gw_arg *Datatype*

Type

```
datatype gw_arg =
      GW_CHILD |
      GW_HWNDFIRST |
      GW_HWNDLAST |
      GW_HWNDNEXT |
      GW_HWNDPREV |
      GW_OWNER
```

gwl_value *Datatype*

Type

```
datatype gwl_value =
      DWL_DLGPROC |
      DWL_MSGRESULT |
      DWL_USER |
     …
```

sb_value *Datatype*

Type

```
datatype sb_value =
      SB_BOTH |
      SB_BOTTOM |
      SB_CTL |
      …
```

esb_value *Datatype*

Type

```
datatype esb_value =
      ESB_DISABLE_BOTH |
      ESB_DISABLE_DOWN |
      ESB_DISABLE_LEFT |
      …
```

sc_value

Datatype

Type

```
datatype sc_value =
      SC_CLOSE
    | SC_CONTEXTHELP
    | SC_DEFAULT
    | SC_HOTKEY
    | SC_HSCROLL
    | SC_KEYMENU
    | SC_MAXIMIZE
    | SC_MINIMIZE
    | SC_MOUSEMENU
    | SC_MOVE
    | SC_NEXTWINDOW
    | SC_PREVWINDOW
    | SC_RESTORE
    | SC_SCREENSAVE
    | SC_SIZE
    | SC_TASKLIST
    | SC_VSCROLL
```

menu_value

Datatype

Type

```
datatype menu_value = SUBMENU of hmenu | ITEM
of word
```

Description

A menu value can either be a normal selectable item or a submenu.

menu_flag

Datatype

Type

```
datatype menu_flag =
      MF_BITMAP |
      MF_BYCOMMAND |
      MF_BYPOSITION |
      …
```

6.6 Type conversion utilities

Conversion functions are provided to allow translation between ML types and C types. This allows values to be passed between the ML functions and the C functions.

For example, a window is implemented as a **`word`** type, so to pass a window in an argument as a **`word`** type requires the function **`windowToWord`**.

To illustrate, the following is an example of how to send a message to get the parent of a window using **`windowToWord`** and **`intToWord`**:

```
sendMessage (getParent mywindow,
             WM_COMMAND,
             WPARAM (intToWord someInt),
             LPARAM (windowToWord mywindow));
```

windowToWord *Function*

Type `val windowToWord : hwnd -> word`

Description Converts a value of type **`hwnd`** to type **`word`**.

messageToWord *Function*

Type `val menuToWord : hmenu -> word`

Description Converts a value of type **`hmenu`** to type **`word`**.

intToWord *Function*

Type `val intToWord : int -> word`

Description Converts a value of type **`int`** to type **`word`**.

nullWord *Value*

Type `val nullWord : word`

Description This is a null word.

wordToString *Function*

Type `val wordToString : word -> string`

Description Converts a value of type **`word`** to type **`string`**.

wordToInt *Function*

Type `val wordToInt : word -> int`

Description Converts a value of type **`word`** to type **`int`**.

wordToSignedInt *Function*

Type `val wordToSignedInt : word -> int`

Description Converts a value of type **`word`** to signed integers.

hiword *Function*

Type `val hiword : word -> int`

Description Converts a value of type **`word`** to type **`int`**.

loword

Function

Type `val loword : word -> int`

Description Converts a value of type **word** to type **int**.

nullWindow

Value

Type `val nullWindow : hwnd`

Description This is a null window.

isNullWindow

Function

Type `val isNullWindow : hwnd -> bool`

Description Returns **true** if the window **hwnd** is a null window.

mainLoop

Function

Type `val mainLoop : unit -> unit`

Description Sets up a Windows event loop for handling user interaction in an application. That is, it processes Windows messages sent to the application.

mainInit

Function

Type `val mainInit : unit -> hwnd`

Description Initializes the user’s application.

doInput *Function*

Type `val doInput : unit -> bool`

Description Processes one Windows message and returns **`true`** if the message was a quit (**`WM_QUIT`**) message.

convertSbValue *Function*

Type `val convertSbValue : sb_value -> int`

Description Converts a value of type **`sb_value`** to type **`int`**.

convertScValue *Function*

Type `val convertScValue : sc_value -> int`

Description Converts a value of type **`sc_value`** to type **`int`**.

newControlId *Function*

Type `val newControlId : unit -> word`

Description Assigns a unique number to the next control to be created, and is used in referencing that control in, for example, adding a command handler for that control.

6.7 Identifier reference

The tables list ML function identifiers and the C functions from the Windows SDK to which they correspond.

Some Windows functions specify the success or failure of their execution by returning a **`BOOL`**, **`int`**, or **`DWORD`**. The equivalent ML functions typically handle failure by raising an exception. As a result such ML functions return a **`unit`**, relying on the exception to clarify the nature of the failure. The functions to which this applies are marked as follows:

- Functions marked § return a **`unit`** as opposed to **`BOOL`** in the Windows equivalent.
- Functions marked † return a **`unit`** as opposed to **`int`** in the Windows equivalent.
- Functions marked ‡ return a **`unit`** as opposed to **`DWORD`** in the Windows equivalent.
- In cases where a Windows function passes a pointer to a structure, the corresponding ML function usually passes the structure itself. Such functions are marked *.

6.7.1 Windows functions

Table 6.2 ML window functions

ML function	C function	Notes
`anyPopup`	**`AnyPopup`**	
`bringWindowToTop`	**`BringWindowToTop`**	
`childWindowFromPoint`	**`ChildWindowFromPoint`**	See page 186.
`closeWindow`	**`CloseWindow`**	
`createWindow`	**`CreateWindow`**	See page 186.
`createWindowEx`	**`CreateWindow`**	See page 187.
`destroyWindow`	**`DestroyWindow`**	§
`enumChildWindows`	**`EnumChildWindows`**	See page 187.
`enumWindows`	**`EnumWindows`**	§

Table 6.2 ML window functions

ML function	C function	Notes
`findWindow`	`FindWindow`	
`getClientRect`	`GetClientRect`	*
`getDesktopWindow`	`GetDesktopWindow`	
`getForegroundWindow`	`GetForegroundWindow`	
`getLastActivePopup`	`GetLastActivePopup`	
`getNextWindow`	`GetNextWindow`	
`getParent`	`GetParent`	
`getTopWindow`	`GetTopWindow`	
`getWindow`	`GetWindow`	
`getWindowRect`	`GetWindowRect`	*
`isChild`	`IsChild`	
`isIconic`	`IsIconic`	
`isWindow`	`IsWindow`	
`isWindowUnicode`	`IsWindowUnicode`	
`isWindowVisible`	`IsWindowVisible`	
`moveWindow`	`MoveWindow`	§
`setForegroundWindow`	`SetForegroundWindow`	§
`setParent`	`SetParent`	
`setWindowText`	`SetWindowText`	§
`showOwnedPopups`	`ShowOwnedPopups`	§
`showWindow`	`ShowWindow`	§
`updateWindow`	`UpdateWindow`	§

Table 6.2 ML window functions

ML function	C function	Notes
windowFromPoint	**WindowFromPoint**	See page 187.

childWindowFromPoint *Function*

The ML function **childWindowFromPoint** corresponds to the C function **ChildWindowFromPoint** in the Windows SDK. It has the following type:

```
val childWindowFromPoint : hwnd * (int * int) -> hwnd
```

The **childWindowFromPoint** function takes the pair **(int * int)** in a manner emulating the Windows structure **POINT**.

createWindow *Function*

The ML function **createWindow** corresponds to the C function **Createwindow** in the Windows SDK. It has the following type:

```
val createWindow : {class : string,
                   name : string,
                   styles : window_style list,
                   width : int,
                   height : int,
                   parent : hwnd,
                   menu : word} -> hwnd
```

The **createWindow** function leaves out the parameters to **CreateWindow** that specify the *x* and *y* coordinates of the window, the handle to the application instance, and the pointer to the window-creation data.

createWindowEx *Function*

The **createWindowEx** function is similar to the **createWindow** function, but includes *x* and *y* coordinate information and dialog style information.

```
val createWindow : {class : string,
                    name : string,
                    styles : window_style list,
                    width : int,
                    height : int,
                    parent : hwnd,
                    menu : word,
                    x : int,
                    y : int,
                    ex_styles : window_ex_style} -> hwnd
```

The **window_ex_style** datatype is a new datatype with the following definition:

```
datatype window_ex_style =
  WS_EX_DLGMODALFRAME |
  WS_EX_STATICEDGE |
  WS_EX_WINDOWEDGE
```

enumChildWindows *Function*

The ML function **enumChildWindows** corresponds to the C function **EnumChildWindows** in the Windows SDK. It has the following type:

```
val enumChildwindows : hwnd * (hwnd * unit) -> unit
```

The function **enumChildWindows** takes two parameters: the window in question and a callback function of type **hwnd -> unit**. It returns a **unit** as opposed to **BOOL** in the windows equivalent.

windowFromPoint *Function*

The ML function **windowFromPoint** corresponds to the C function **WindowFromPoint** in the Windows SDK. It has the following type:

```
val windowFromPoint : int * int -> hwnd
```

The **windowFromPoint** function takes the pair **(int * int)** in a manner emulating the Windows structure **POINT**.

6.7.2 Messages

Table 6.3 ML message functions

ML function	C function	Notes
getInputState	**GetInputState**	
getMessagePos	**GetMessagePos**	See page 188.
getMessageTime	**GetMessageTime**	See page 189.
inSendMessage	**InSendMessage**	
postMessage	**PostMessage**	§
postQuitMessage	**PostQuitMessage**	§
sendMessage	**SendMessage**	

getMessagePos *Function*

The ML function **getMessagePos** corresponds to the C function **GetMessagePos** in the Windows SDK. It has the following type:

```
val getMessagePos : unit -> int * int
```

The ML function returns **int * int** in a manner corresponding to the Windows function, which returns **DWORD**.

getMessageTime *Function*

The ML function **`getMessageTime`** corresponds to the C function **`GetMessageTime`** in the Windows SDK. It has the following type:

```
val getMessageTime : unit -> int
```

The ML function returns **`int`** in a manner corresponding to the Windows function, which returns **`LONG`**.

6.7.3 Window classes

Table 6.4 ML window classes

ML function	C function	Notes
`enableWindow`	**`EnableWindow`**	
`setWindowLong`	**`SetWindowLong`**	

6.7.4 Keyboard input

Table 6.5 ML keyboard input functions

ML function	C function	Notes
`enablewindow`	**`EnableWindow`**	
`getActiveWindow`	**`GetActiveWindow`**	
`getFocus`	**`GetFocus`**	
`isWindowEnabled`	**`IsWindowEnabled`**	
`setActiveWindow`	**`SetActiveWindow`**	
`setFocus`	**`SetFocus`**	

6.7.5 Mouse input

Table 6.6 ML mouse input functions

ML function	C function	Notes
getCapture	**GetCapture**	
releaseCapture	**ReleaseCapture**	§
setCapture	**SetCapture**	

6.7.6 Timers

Table 6.7 ML timer functions

ML function	C function	Notes
killTimer	**KillTimer**	§
setTimer	**SetTimer**	

6.7.7 Buttons

The word types here refer to the button control identifiers. The **hwnd** type refers to the window handle of the dialog box.

Table 6.8 ML button functions

ML function	C function	Notes
checkDlgButton	**CheckDlgButton**	§
checkRadioButton	**CheckRadioButton**	§

Table 6.8 ML button functions

ML function	C function	Notes
isDlgButtonChecked	**IsDlgButtonChecked**	

6.7.8 Scroll bars

Table 6.9 ML scroll bar functions

ML function	C function	Notes
enableScrollBar	**EnableScrollBar**	§
getScrollPos	**GetScrollPos**	
getScrollRange	**GetScrollRange**	*§
setScrollPos	**SetScrollPos**	†
setScrollRange	**SetScrollRange**	§
showScrollBar	**ShowScrollBar**	§
getScrollInfo	**GetScrollInfo**	See page 191.

getScrollInfo *Function*

The ML function **getScrollInfo** corresponds to the C function **GetScrollInfo** in the Windows SDK. It has the following type:

```
val getScrollInfo : hwnd * sb_value ->
                    int * word *word * int * int * word * int * int
```

6.7.9 Menus

Table 6.10 ML menu functions

ML function	C function	Notes
appendMenu	**AppendMenu**	§
checkMenuItem	**CheckMenuItem**	‡
createMenu	**CreateMenu**	
createPopupMenu	**CreatePopupMenu**	
destroyMenu	**DestroyMenu**	§
deleteMenu	**DeleteMenu**	§
drawMenuBar	**DrawMenuBar**	§
enableMenuItem	**EnableMenuItem**	§
getMenu	**GetMenu**	
getMenuItemId	**GetMenuItemId**	
getMenuItemCount	**GetMenuItemCount**	
getMenuState	**GetMenuState**	
getMenuString	**GetMenuString**	see page 193.
getSubmenu	**GetSubmenu**	
getSystemMenu	**GetSystemMenu**	
setMenu	**SetMenu**	§
removeMenu	**RemoveMenu**	§

getMenuString *Function*

The ML function **`getMenuString`** corresponds to the C function **`GetMenuString`** in the Windows SDK. It has the following type:

```
val getMenuString : hmenu * word * menu_flag -> string
```

The ML function passes the result as a **`string`**, whereas the Windows function moves the string to the location pointed to by the pointer parameter. The Windows function also includes a parameter for the maximum length of the string, whereas the ML function does not.,

6.7.10 Keyboard accelerators

accelerator_flag *Datatype*

```
datatype accelerator_flag =
      FALT
    | FCONTROL
    | FNOINVERT
    | FSHIFT
    | FVIRTKEY
```

Table 6.11 ML keyboard accelerator functions

ML function	C function	Notes
`createAcceleratorTable`	**`CreateAcceleratorTable`**	See page 194.
`destroyAcceleratorTable`	**`DestroyAcceleratorTable`**	§

createAcceleratorTable *Function*

The ML function **createAcceleratorTable** corresponds to the C function **CreateAcceleratorTable** in the Windows SDK. It has the following type:

```
val createAcceleratorTable : (accelerator_flag list * in * int)
                             list -> accelerator_table
```

The first **int** type parameter is the key, the second is the command identifier.

6.7.11 Dialog boxes

message_box_style *Function*

```
datatype message_box_style =
      MB_ABORTRETRYIGNORE |
      MB_APPLMODAL |
      MB_ICONASTERISK |
      MB_ICONEXCLAMATION |
      MB_ICONHAND |
      MB_ICONINFORMATION |
      MB_ICONQUESTION |
      MB_ICONSTOP |
      MB_OK |
      MB_OKCANCEL |
      MB_RETRYCANCEL |
      MB_YESNO |
      MB_YESNOCANCEL
```

Table 6.12 ML dialog box functions

ML function	C function	Notes
messageBox	**MessageBox**	
endDialog	**EndDialog**	§

Table 6.12 ML dialog box functions

ML function	C function	Notes
getDlgItem	**GetDlgItem**	
getDialogBaseUnits	**GetDialogBaseUnits**	

6.7.12 Painting and drawing

rop2_mode *Datatype*

```
datatype rop2_mode =
      R2_BLACK |
      R2_COPYPEN |
      R2_MASKNOTPEN |
      R2_MASKPEN |
      R2_MASKPENNOT |
      R2_MERGENOTPEN |
      R2_MERGEPEN |
      R2_MERGEPENNOT |
      R2_NOP |
      R2_NOT |
      R2_NOTCOPYPEN |
      R2_NOTMASKPEN |
      R2_NOTMERGEPEN |
      R2_NOTXORPEN |
      R2_WHITE |
      R2_XORPEN
```

rop_mode *Datatype*

```
datatype rop_mode =
      BLACKNESS |
      DSTINVERT |
      MERGECOPY |
      MERGEPAINT |
      NOTSRCCOPY |
      NOTSRCERASE |
      PATCOPY |
      PATINVERT |
      PATPAINT |
      SRCAND |
      SRCCOPY |
      SRCERASE |
      SRCINVERT |
      SRCPAINT |
      WHITENESS
```

Table 6.13 ML painting and drawing functions

ML function	C function	Notes
getBkColor	**GetBkColor**	
setBkColor	**SetBkColor**	
validateRect	**ValidateRect**	See page 197.
invalidateRect	**InvalidateRect**	See page 197.
windowFromDC	**WindowFromDC**	
setPixel	**SetPixel**	
getRop2	**GetROP2**	
setRop2	**SetROP2**	

validateRect *Function*

The ML function **`validateRect`** corresponds to the C function **`ValidateRect`** in the Windows SDK. It has the following type:

```
val validateRect : hwnd * rect MLWorks.Option.option -> unit
```

Whereas the Windows function returns a **`BOOL`**, the ML function returns **`unit`**, and raises an exception if the function fails. Furthermore, the ML function passes the actual **`rect`** structure, whereas the Windows function passes a pointer.

invalidateRect *Function*

The ML function **`invalidateRect`** corresponds to the C function **`InalidateRect`** in the Windows SDK. It has the following type:

```
val invalidateRect : hwnd * rect MLWorks.Option.option * bool
                     -> unit
```

Whereas the Windows function returns a **`BOOL`**, the ML function returns **`unit`**, and raises an exception if the function fails. Furthermore, the ML function passes the actual **`rect`** structure, whereas the Windows function passes a pointer.

6.7.13 Cursors

Table 6.14 ML cursor functions

ML function	C function	Notes
`clipCursor`	**`ClipCursor`**	§
`getClipCursor`	**`GetClipCursor`**	*
`getCursorPos`	**`GetCursorPos`**	*
`setCursorPos`	**`SetCursorPos`**	§

Table 6.14 ML cursor functions

ML function	C function	Notes
`showCursor`	**`ShowCursor`**	

6.7.14 Clipboard

Table 6.15 ML clipboard functions

ML function	C function	Notes
`openClipboard`	**`OpenClipboard`**	
`closeClipboard`	**`CloseClipboard`**	§
`emptyClipboard`	**`EmptyClipboard`**	*§
`setClipboardData`	**`SetClipboardData`**	*
`getClipboardData`	**`GetClipboardData`**	

setClipboardData *Function*

The ML function **`setClipboardData`** corresponds to the C function **`SetClipboardData`** in the Windows SDK. It has the following type:

```
val setClipboardData : string -> unit
```

Instead of passing a data handle like the corresponding Windows function, the ML function **`setClipboardData`** passes the actual data string.

6.7.15 Device contexts

object
Datatype

```
datatype object = OBJECT of word
```

hbrush
Datatype

```
datatype hbrush = HBRUSH of word
```

hpen
Datatype

```
datatype hpen = HPEN of word
```

object_type
Datatype

```
datatype object_type =
      OBJ_PEN |
      OBJ_BRUSH |
      OBJ_PAL |
      OBJ_FONT |
      OBJ_BITMAP
```

stock_object *Datatype*

```
datatype stock_object =
      ANSI_FIXED_FONT |
      ANSI_VAR_FONT |
      BLACK_BRUSH |
      BLACK_PEN |
      DEFAULT_GUI_FONT |
      DEFAULT_PALETTE |
      DKGRAY_BRUSH |
      GRAY_BRUSH |
      HOLLOW_BRUSH |
      LTGRAY_BRUSH |
      NULL_BRUSH |
      NULL_PEN |
      OEM_FIXED_FONT |
      SYSTEM_FIXED_FONT |
      SYSTEM_FONT |
      WHITE_BRUSH |
      WHITE_PEN
```

Table 6.16 ML device context functions

ML function	C function	Notes
cancelDC	**CancelDC**	
createCompatibleDC	**CreateCompatibleDC**	
deleteObject	**DeleteObject**	§
getCurrentObject	**GetCurrentObject**	
getDC	**GetDC**	
getDCOrgEx	**GetDCOrgEx**	
getStockObject	**GetStockObject**	
releaseDC	**ReleaseDC**	†
restoreDC	**RestoreDC**	§
saveDC	**SaveDC**	

Table 6.16 ML device context functions

ML function	C function	Notes
selectObject	**SelectObject**	

6.7.16 Bitmaps

Table 6.17 ML bitmap function

ML function	C function	Notes
bitBlt	**bitBlt**	§

bitBlt *Function*

```
val bitBlt :
   {hdcDest: hdc,
  hdcSrc: hdc,
  height: int,
  ropMode: rop_mode,
  width: int,
  xDest: int,
  xSrc: int,
  yDest: int,
  ySrc: int} -> unit
```

6.7.17 Brushes

hatch_style *Datatype*

```
datatype hatch_style =
     HS_BDIAGONAL
    | HS_CROSS
    | HS_DIAGCROSS
    | HS_FDIAGONAL
    | HS_HORIZONTAL
    | HS_VERTICAL
```

Table 6.18 ML brushes functions

ML function	C function	Notes
createHatchBrush	**CreateHatchBrush**	
createSolidBrush	**CreateSolidBrush**	

6.7.18 Pens

pen_style *Datatype*

```
datatype pen_style =
      PS_DASH |
      PS_DASHDOT |
      PS_DASHDOTDOT |
      PS_DOT |
      PS_NULL |
      PS_SOLID |
      PS_INSIDEFRAME
```

Table 6.19 ML pen function

ML function	C function	Notes
createPen	**CreatePen**	

6.7.19 Lines and curves

arc_direction *Datatype*

```
datatype arc_direction = AD_COUNTERCLOCKWISE | AD_CLOCKWISE
```

Table 6.20 ML lines and curves functions

ML function	C function	Notes
angleArc	**AngleArc**	§
arc	**Arc**	§
arcTo	**ArcTo**	§
getArcDirection	**GetArcDirection**	
lineTo	**LineTo**	§
moveTo	**MoveTo**	§
polyBezier	**PolyBezier**	§
polyBezierTo	**PolyBezierTo**	§
polyline	**Polyline**	§
polylineTo	**PolylineTo**	§
polyPolyline	**PolyPolyline**	§

Table 6.20 ML lines and curves functions

ML function	C function	Notes
`setArcDirection`	**`SetArcDirection`**	†

6.7.20 Filled Shapes

Table 6.21 ML filled shapes functions

ML function	C function	Notes
`chord`	**`Chord`**	§
`ellipse`	**`Ellipse`**	§
`fillRect`	**`FillRect`**	†
`frameRect`	**`FrameRect`**	†
`invertRect`	**`InvertRect`**	§
`pie`	**`Pie`**	§
`polygon`	**`Polygon`**	§
`polyPolygon`	**`PolyPolygon`**	§
`rectangle`	**`Rectangle`**	§
`roundRect`	**`RoundRect`**	§

6.7.21 Fonts and text

Table 6.22 ML font and text functions

ML function	C function	Notes
getTextColor	**GetTextColor**	
getTextExtentPoint	**GetTextExtentPoint**	See page 205.
setTextColor	**SetTextColor**	
textOut	**TextOut**	*§

getTextExtentPoint *Function*

The ML function **getTextExtentPoint** corresponds to the C function **GetTextExtentPoint** in the Windows SDK. It has the following type:

```
val getTextExtentPoint : hdc * string -> int * int
```

Instead of returning a **BOOL** like the Windows function, the ML function returns **int * int** for the width and height of the text, and raises a **WindowSystemError** exception on error.

6.7.22 Coordinate spaces and transformations

Table 6.23 ML coordinate functions

ML function	C function	Notes
clientToScreen	**ClientToScreen**	*
screenToClient	**ScreenToClient**	*

6.7.23 Sound

Table 6.24 ML sound function

ML function	C function	Notes
messageBeep	**MessageBeep**	§

6.7.24 System information

color_spec *Datatype*

```
datatype color_spec =
      COLOR_ACTIVEBORDER |
      COLOR_ACTIVECAPTION |
      COLOR_APPWORKSPACE |
      COLOR_BACKGROUND |
      COLOR_BTNSHADOW |
      COLOR_BTNTEXT |
      COLOR_CAPTIONTEXT |
      COLOR_GRAYTEXT |
      COLOR_HIGHLIGHT |
      COLOR_HIGHLIGHTTEXT |
      COLOR_INACTIVEBORDER |
      COLOR_INACTIVECAPTION |
      COLOR_INACTIVECAPTIONTEXT |
      COLOR_MENU |
      COLOR_SCROLLBAR |
      COLOR_WINDOW |
      COLOR_WINDOWFRAME |
      COLOR_WINDOWTEXT
```

Table 6.25 ML system information functions

ML function	C function	Notes
getSysColor	**GetSysColor**	
openFileDialog	**OpenFileDialog**	
openDirDialog	**OpenDirDialog**	
saveAsDialog	**SaveAsDialoG**	
saveImageDialog	**SaveImageDialog**	

6.7.25 MLWorks-specific functions

setAcceleratorTable *Function*

The ML function **setAcceleratorTable** corresponds to the Windows function **CopyAcceleratorTable**. It has the following type:

```
val setAcceleratorTable : accelerator_table -> unit
```

The function **setAcceleratorTable** preforms the equivalent action that the Windows command **CopyAcceleratorTable(table,NULL,00**, and the resulting table is stored for future reference.

registerPopupWindow *Function*

```
val registerPopupWindow : hwnd -> unit
```

This function registers the popup window within the window manager, allowing the user to use the **Tab** or cursor keys to navigate into and out of the new popup window.

unregisterPopupWindow *Function*

```
val unregisterPopupWindow : hwnd -> unit
```

This function unregisters a window registered using **registerPopupWindow**.

6.7.26 Window procedures

addMessageHandler *Function*

```
val addMessageHandler : hwnd * message *
(wparam * lparam -> word MLWorks.Option.option) -> unit
```

The function **addMessageHandler** takes arguments **window**, **message**, and **handler** — a function returning an option value. The function **handler** is called whenever **window** receives a message of type **message**.

addNewWindow *Function*

```
val addNewWindow : hwnd * word -> unit
```

This function takes a window and a C procedure and adds the pair to the **handler_map_entry** list.

removeWindow *Function*

```
val removeWindow : hwnd -> unit
```

This function takes a window and removes all associated C procedures from the **handler_map_entry** list.

getMlWindowProc *Function*

```
val getMlWindowProc : unit -> word
```

This function returns the Windows procedure for the given window. See the Windows SDK documentation for details on Windows procedures.

addCommandHandler *Function*

```
val addCommandHandler : hwnd * word * (hwnd * int -> unit) -> unit
```

A window can receive the **WM_COMMAND** message which has a list of other messages associated with it. The function **addCommandHandler** adds a handler function for this type of message to the given control on the given window. Typically this handler then deals with the messages that are received as part of the **WM_COMMAND** message. The ML function returns a **unit** instead of an option value.

6.7.27 Miscellaneous

malloc *Function*

```
val malloc : int -> word
```

Allocates a specified amount of memory and returns its address.

free *Function*

```
val free : word -> unit
```

Frees the memory associated with the given address.

setByte *Function*

```
val setByte : word * int * int -> unit
```

The **setByte** function takes three arguments, **Address**, **Offset**, and **Value**, and sets the value at the address given by **(Address + Offset)** to **Value**, where **Value** is a byte or character.

makeCString *Function*

```
val makeCString : string -> word
```

Takes an ML string and returns the memory address of the string.

7

The MLWorks Foreign Interface Library

7.1 Introduction

This MLWorks foreign interface library (FI, for short) is provided in the **`Interface`** structure. It is not built in to the MLWorks interactive environment, but is provided as a set of source and separately compiled files. See the installation notes for your version of MLWorks for details of their location.

The MLWorks foreign interface library provides facilities for interfacing ML applications to code written in C. The design accommodates other languages, and we may add support for them in future.

Note: The MLWorks foreign interface library is currently being revised and redesigned in future versions of MLWorks.

To use this library in your applications, add

```
require "$.foreign.__interface";
```

to the start of the relevant source files. To load the library into the MLWorks interactive environment, use the project system or the load functions in **`Shell.Project`**.

7.1.1 Terminology

In general, all ML expressions evaluate to *values*. An ML *object* is simply an ML value which is also *stateful* and so has persistent effect. ML computes values which can be either stateful or *stateless*. We shall call ML values that are stateless *pure* ML values.

In ML programming work generally, transient values tend not to be stateful. Objects are used mainly when information persists from transaction to transaction. The programming model required for using the MLWorks FI is usually procedural and imperative, relying very much upon persistent state.

This situation is almost forced, because most foreign languages are to some degree or even entirely static and imperative. In particular, function calling requires addresses and pointers to be consistent during a foreign function call. This can be achieved by providing static memory and having ML operators construct and analyse data in these static areas.

We have not followed convention and called the FI a Foreign Function Interface or FFI. The reason for this is that ML is strongly typed, and so the interface must also be concerned with the differences in typing between different languages. Further, the concept of function is central to ML and is certainly not neutral from an ML perspective. For this reason, we chose to call our interface a Foreign Interface: a term which is simpler and which implies greater generality than Foreign Function Interface.

7.2 Overview of the FI

This section provides an overview of using the FI, explaining how C code can be interfaced to ML code. There is a small example given to show some of the key features of the FI. The code for the example is found in the MLWorks installation subdirectory **`foreign/samples/`**. The example files are:

`hello.c`	Example C file
`hello.sml`	Example ML file containing FI calls for interfacing to **`hello.c`**.
`Makefile`	A suitable makefile for your platform.

In addition, **`libml.h`**, a C header file, which must be included in any C programs which needs to access ML data or call ML functions, is available under the **bin** subdirectory of the top-level MLWorks installation directory.

7.2.1 Loading the FI into MLWorks

Load the FI into the MLWorks interactive environment by entering the following in a listener:

```
Shell.Project.load "$.foreign.__interface";
```

7.2.2 On the C side ...

The following short piece of C code is in **hello.c**:

```
#include <stdio.h>

int hello(char *str, int num)
    {
      printf("%s %i\n", str, num);
      return(42 + num);
    }
```

Note that **hello.c** does not contain a **main** function — it is instead providing a shared library, albeit a trivial one.

Compile the **hello.c** file using your favourite C compiler to create an object file, and from that a shared object file. A sample makefile to do this might look like this (on Solaris 2.5):

```
hello.o: hello.c
  $(CC) $(CFLAGS) -c hello.c -o hello.o

hello.so: hello.o
  $(LD) -Bdynamic -G -lgen hello.o -o hello.so
```

The variants for Linux and IRIX are fairly trivial. A suitable makefile for the platform on which you are running MLWorks is available in the Foreign Interface distribution under the directory **foreign/samples/**.

To make use of UNIX shared libraries from MLWorks (and indeed in general) it is *very* important to set the **LD_LIBRARY_PATH** environment variable appropriately. For correct operation, the path *must* include the current directory (**.**) and the standard shared-object systems directory (usually **/usr/lib**). This path is very sensitive to ordering, so if you have difficulty with it, experiment with different orderings and do not rely on the documented defaults.

7.2.3 On the ML side...

Having constructed a shared library in C, we want to make use of it from ML. To do this within MLWorks, we use the tools provided by the **`Interface`** structure.

Note: For ease of presentation, we assume that the **`Interface`** structure and its substructures have all been opened with **`open`**. However, we do not advise the use of **`open`** in programs because it makes them difficult to debug.

The ML side of the interface now goes as follows; note that system responses will be prefixed by the greater-than (**`>`**) sign. All of the ML code here is available in **`foreign/samples/hello.sml`**.

The first step is to the foreign code itself. This action creates an ML object called a **`c_structure`**. For example:

```
val hello_struct =
 Interface.C.Structure.loadObjectFile("hello.so",
Interface.C.Structure.IMMEDIATE_LOAD);
> val hello_struct : c_structure = _
```

Once this action is complete, raw foreign code will have been loaded into MLWorks. We now want to feed data as arguments to functions in the foreign code — in this case, the **`hello`** function in the C program — and then to accept the results of the foreign computations as they are returned to ML. However, there is no means of accessing the foreign code from ML yet.

To access the foreign code we must build ML entities which can access and manipulate foreign data. The FI provides a range of features to help us do so.

The FI requires that foreign data be associated with an ML type. Moreover, there must also be some way of describing how foreign data is to be interpreted and, as it were, “understood”. From an operational point of view, this understanding amounts to a qualification of what operations the foreign data may participate in, and hence what form that participation could take.

So, in the FI, each piece of foreign data comes equipped with a *certificate* which describes what the interpretation of the data is at any given time. Since these certificates bear information, they must themselves be represented in terms of ML values.

However, the FI further separates the data storage of data values from their representation, by mapping the actual values into “workspace” objects called *stores*. The interpretation of these data values is then contained in another kind of ML entity called a **c_object**, which is rather like a disembodied “container”. The idea is that objects are generally associated with a place within some store containing the object’s data value. This indirection between object value and the interpretation of that value provides considerable flexibility, even within a strongly typed framework such as ML. Such flexibility is necessary for mimicing enough of a foreign data-typing scheme.

So the next step is to build a store object which will contain the data values, such as the arguments to, and the results from, foreign calls. For example:

```
val hello_store =Interface.Store.store{alloc =
Interface.Store.ALIGNED_4, overflow = Interface.Store.BREAK, size
= 60, status = Interface.Store.RDWR_STATUS };
> val hello_store : store = _
```

We now have a workspace for storing data values relating to the foreign code. The parameters in the call above say that:

- Allocation is 4-byte aligned.
- An exception is raised if too much memory is requested.
- The store is 60 bytes in size.
- The store permits both reading and writing of data.

The next step is to build some objects through which foreign data can be manipulated and accessed:

```
val void_object = Interface.C.Value.object { ctype =
Interface.C.Type.VOID_TYPE, store = hello_store };
> val void_object : c_type object = _

val str_object = Interface.C.Value.object { ctype =
Interface.C.Type.STRING_TYPE{ length = 30 }, store = hello_store
};
> val str_object : c_type object = _
```

```
val int_object1 = Interface.C.Value.object { ctype =
Interface.C.Type.INT_TYPE, store = hello_store };
> val int_object1 : c_type object = _

val int_object2 = Interface.C.Value.object { ctype =
Interface.C.Type.INT_TYPE, store = hello_store };
> val int_object2 : c_type object = _

val ptr_object = Interface.C.Value.object { ctype =
Interface.C.Type.ptrType(Interface.C.Type.VOID_TYPE), store =
hello_store };
> val ptr_object : c_type object = _
```

These objects are associated with particular places in the **hello_store**. However, because no values have yet been bound to these objects, read operations upon them are assumed to be invalid. Once they have been written to, they can then be read safely. The following code initializes some of these objects:

```
Interface.C.Value.setString(str_object, "What is 65 - 42? ----
Ans is ");
Interface.C.Value.setInt(int_object1, 23);
```

The following code extracts their values:

```
Interface.C.Value.getString(str_object);
> val it : string = "What is 65 - 42? ---- Ans is "

Interface.C.Value.getInt(int_object1);
> val it : int = 23
```

Having set this data up, we need to use it in conjunction with the foreign code we have already loaded. To do this, we need some signature information concerning the foreign functions we want to use. This requires an empty **c_signature** object:

```
val hello_sig = Interface.C.Signature.newSignature();
> val hello_sig : c_signature = _
```

Next add the following entry to the signature. The entry corresponds to the foreign function we wish to use:

```
Interface.C.Signature.defEntry(hello_sig,
Interface.C.Signature.FUN_DECL { name = "hello",
  source = [Interface.C.Type.ptrType(Interface.C.Type.CHAR_TYPE),
  Interface.C.Type.INT_TYPE] : Interface.C.Type.c_type list,
  target = Interface.C.Type.INT_TYPE }
);
```

Note how the form of the signature information follows the structure of the ANSI C prototype for the function.

We can now use the **c_signature** and **c_structure** information we have obtained to extract *callable entries* for the foreign functions they provide.

```
val def_hello =
Interface.C.Function.defineForeignFun(hello_struct, hello_sig);
> val def_hello : filename -> c_function = fn
```

Using this, we can obtain a **c_function** object that can then be called directly:

```
val hello = def_hello "hello";
> val hello : c_function = _
```

The above allows foreign functions to be extracted as ML values and bound to ML identifiers in the usual way.

We have almost reached the point at which we can call our foreign function. Before we do, we need to set up the first argument as a character pointer to some string data:

```
Interface.C.Value.setPtrType { ptr = ptr_object, data =
str_object };
Interface.C.Value.setPtrAddrOf { ptr = ptr_object, data =
str_object };
Interface.C.Value.castPtrType { ptr = ptr_object, ctype =
Interface.C.Type.CHAR_TYPE };
```

First, the pointer was set to the appropriate type, **str_object**. Then it was set to point at the **str_object** data. Finally, the pointer was cast to the required argument type. In fact, because strings are such a frequent case, the FI can accept **STRING_TYPE** values directly and convert them into an appropriate **CHAR_TYPE** pointer, for both argument and result from a foreign function.

Finally, we can call our foreign function **hello()** using all we have put together:

```
Interface.C.Function.call hello ([ptr_object,int_object1],
int_object2);
> val it : unit = ()
```

The above call required two objects to give the argument values and an object to accept the result. The string

```
> What is 65 - 42? ---- Ans is 23
```

is printed to the standard output.

After the call the result value is deposited in **`int_object2`**. We can extract this value with the following:

```
getInt(int_object2);
> val it : int = 65
```

7.2.4 Summary

By means of a brief example we have shown how the FI provides a data model that is consistent with C. However, the FI provides an architecture supporting foreign languages in general. It has a general component, common to all languages supported, and allows for specific components dealing with the language in question.

7.3 Some limitations and future extensions

The main limitation at present is that the foreign function call is not completely general. Both arguments and results are limited to being values of size at most 4 bytes. Such values can be:

- characters
- standard integers (short, long, signed or unsigned)
- simple floats (not doubles)
- enumerated constants
- machine pointers to general values (including structures and functions)

The final case allows general strings to be handled and general data to be used. In practice, this is only a limitation when a general **`struct`** value or a **`double`** needs to be passed or returned directly.

7.4 The top-level structure: Interface

The top-level structure for the FI **`Interface`**. It contains the following substructures:

```
structure Store
structure Object
structure Aliens
structure LibML
structure Diagnostic
structure C
structure C.Structure
structure C.Type
structure C.Value
structure C.Signature
structure C.Function
structure C.Diagnostic
```

In addition to these structures, various standard types used throughout the interface are exported. They are:

`type word32` A standard 32-bit value. Equivalent to **`Word32.word`**.

`type address` Address type is 32-bit addresses. Equivalent to **`word32`**.

`type bytearray` Byte array values are supplied using standard MLWorks byte arrays: **`MLWorks.ByteArray.bytearray`**.

`type name` Names are standard ML strings.

`type filename` Filenames are standard ML strings.

The following sections document each of the **`Interface`** sub-structures.

7.5 The Store structure

The **`Store`** structure defines store objects and operations upon them. The idea behind stores is straightforward: they provide the underlying workspace in which foreign data is represented. To access and manipulate foreign data, you must declare objects associated with particular locations within stores. It is through these objects that foreign data can be read and written. Store objects can be relocated under user control within their associated store.

A store therefore represents a statically allocated, uniformly addressable (that is, contiguous) workspace, in which interfacing can take place as a direct action upon memory. Stores have additional structure to make them more robust and convenient for programming. For example, you have control over what happens if and when a store overflows. A possible overflow policy is to raise an exception; another policy automatically expands the store and increases the workspace available.

Stores are not specific to any particular language interface.

7.5.1 Machine pointers and stores

Much foreign data can consist of pointers, represented by explicit machine addresses. Clearly such data may be literally represented within an ML store. However, as noted above, stores can expand in size. To ensure the uniformity of addressing, this expansion is implemented by copying the data in them. Unfortunately this copying invalidates any explicit pointers to other data elements contained in the same store. Other explicit pointers referring to non-local (or remote) data naturally remain valid.

Explicit machine pointers must therefore be treated with care. For example, there is a facility for providing 'local' pointers, which are represented as a local offset from the base address of the store. The use of small indices here means that conventional array indexing can be used directly from ML. Of course, this facility also requires the ability to convert between local indices and actual machine addresses.

The advantage of using 'local' pointers is that they remain invariant under expansion of the store. However, their disadvantage is that they are meaningless if used out of context. Hence, local pointers must not be passed into foreign code — they must first be converted into machine addresses. The FI provides tools for performing these conversions.

One way of avoiding these difficulties is to work with stores that cannot be expanded, but which have sufficient static space allocated to start with. In this case, machine addresses cannot be invalidated due to store expansion and so can be passed to foreign code with impunity.

7.5.2 Stores

store *Type abbreviation*

Specification: `type store`

Description:

An encapsulated type representing store objects.

store_status *Datatype*

Specification: **`datatype store_status = LOCKED_STATUS | RD_STATUS | WR_STATUS | RDWR_STATUS`**

Description:

Each store has a status, which can take the following values:

`LOCKED_STATUS` Store data may not be accessed or modified by ML.

`RD_STATUS` Store data is read-only from ML.

`WR_STATUS` Store data is write-only from ML.

`RDWR_STATUS` Store data is readable / writeable from ML (the default).

ReadOnly *Exception*

Specification: **`exception ReadOnly`**

Description:

This exception is raised if an attempt is made to write data to a store whose status forbids writing: either **`LOCKED_STATUS`** or **`RD_STATUS`**.

WriteOnly *Exception*

Specification: **`exception WriteOnly`**

Description:

This exception is raised if an attempt is made to read data from a store whose status forbids reading: either **`LOCKED_STATUS`** or **`WR_STATUS`**.

storeStatus *Function*

Signature: **`val storeStatus : store -> store_status`**

Description:

Function for inspecting the status of a store object.

setStoreStatus *Function*

Signature:

```
val setStoreStatus :
    (store * store_status) -> unit
```

Description:

Function for setting the status of a store object.

alloc_policy *Datatype*

Specification:

```
datatype alloc_policy =
    ORIGIN | SUCC | ALIGNED_4 | ALIGNED_8
```

Description:

A store object is created just like any other ML value (except that it is static, that is, the garbage collector may not relocate it) and given some memory for its representation.

However, a store is involved in managing a number of **`obj`** objects associated with its workspace area. The **`alloc_policy`** datatype is used to specify the manner in which space is given to these **`obj`** objects from within the store's workspace:

`ORIGIN` Newly created objects are initially located at the origin. Once created, such objects may be moved around with their host store by using relocation operations. In this way, you have control of the arrangement of objects within the store.

Object relocation operations are obviously sensitive to the underlying data model of the foreign language, and so are implemented by the language-specific component of the FI.

`SUCC` Each fresh object is located at the 'top' of the workspace, immediately following all the other objects.

`ALIGNED_4` As for **`SUCC`**, but each fresh object is allocated on a 4-byte address boundary (that is, the address is 0 mod 4).

ALIGNED_8 As for **SUCC**, but each fresh object is allocated on a 8-byte address boundary (that is, the address is 0 mod 8).

It is possible to have several stores in use at the same time. Each could have different allocation policies, in order to handle different kinds of data.

overflow_policy *Datatype*

Specification:

```
datatype overflow_policy =
    BREAK | EXTEND | RECYCLE
```

Description:

Each store object in effect manages a piece of workspace memory on ML's behalf, and objects are associated with parts of this workspace. A store is said to have *overflowed* when an attempt is made to use more space than is presently available in the associated workspace. When overflow occurs, an overflow policy is enacted. The **overflow_policy** datatype provides a number of possibilities when overflow occurs:

BREAK The exception **ExpandStore** is raised upon an attempt to expand the store (possibly made with the **expand** function, described on page 226). A store with this overflow policy is effectively fixed in size because it cannot be expanded.

EXTEND The store automatically expands, by amount determined by an internal rule, to accommodate further allocation requests. This expansion is obviously subject to system limits on the amount of memory that a process can have mapped at a time.

Explicit calls to **expand** (see page 226) need advice on how much extra space should be allocated.

In effect, stores with this overflow policy are flexible in size and can be expanded as necessary by automatic or manual methods.

RECYCLE Allocation resumes at the origin of the store, overwriting any data presently at the origin. This policy is suitable for stores containing ephemeral objects, that is, objects whose lifetimes are known in advance to be short.

There is clearly a danger that live data can be overwritten in a store using this policy.

Stores with this overflow policy may be explicitly expanded. If a request to allocate more space cannot be satisfied for some reason, the **ExpandStore** exception is raised.

store *Function*

Signature:

```
val store :
    { alloc : alloc_policy,
      overflow : overflow_policy,
      status : store_status,
      size : int
    } -> store
```

Description:

This function generates fresh stores. The initial size, allocation policy, overflow policy, and initial store status can be supplied.

The **store_status** may be modified using the **setStoreStatus** function (see page 222), and the store's **size** may be explicitly increased (when possible) using the **expand** function (see page 226). The other store attributes cannot be modified dynamically.

storeSize *Function*

Signature: `val storeSize : store -> int`

Description:

This function returns the current size of the store.

storeAlloc *Function*

Signature: **`val storeAlloc : store -> alloc_policy`**

Description:

This function returns the allocation policy for the store.

storeOverflow *Function*

Signature: **`val storeOverflow : store -> overflow_policy`**

Description:

This function returns the overflow policy for the store.

isStandardStore *Function*

Signature: **`val isStandardStore : store -> bool`**

Description:

This predicate determines if the store is considered to be standard. A store is *standard* when the allocation policy is not **`ORIGIN`** or if the overflow policy is not **`RECYCLE`**.

isEphemeralStore *Function*

Signature: **`val isEphemeralStore : store -> bool`**

Description:

This predicate determines if the store is considered to be ephemeral. A store is *ephemeral* when the allocation policy is not **`ORIGIN`** and the overflow policy is **`RECYCLE`**.

ExpandStore *Exception*

Specification: **`exception ExpandStore`**

Description:

This exception is raised when an attempt to expand a store cannot be met.

expand *Function*

Signature: **`val expand : (store * int) -> unit`**

Description:

This function expands a store by at least the specified size (given in bytes), or fails with exception **`ExpandStore`**.

7.6 The Object structure

A foreign object is an ML value which provides a means of both accessing and modifying foreign data from ML. Foreign objects are represented with the type **`object`**.

Foreign objects do not contain foreign data itself, but are instead associated with a location in a store object which contains foreign data. In short, a foreign object provides *indirect* access to foreign data, thus allowing objects to be freely copied or otherwise manipulated without replicating the foreign data itself. The disadvantage of this is that it permits many different objects to refer to the same foreign data, that is, it permits aliasing. An update to the foreign data through any one such object is an update that will be observed by all alias objects.

The requirements for representing foreign objects naturally differ depending on the foreign language being interfaced to. Some of the necessary features of, and possible operations upon, foreign objects will be the same whatever the foreign language.

However, foreign objects will probably have some language-specific aspects too. In particular, any notion of typing will be language specific. For this reason, the ML type that represents **`object`** objects is polymorphic, allowing for this dependence on language-specific aspects, such as typing.

The **`Object`** structure is, then, the generic implementation of foreign object representations and protocols.

As you will see below, there are several generic ways of inspecting a foreign object, but *no* (generic) ways of generating objects directly or modifying any existing characteristics they may have. The reason is that these are very much subject to the language-specific semantics of the appropriate data model. Accordingly, such operations are provided within the language-specific interfaces.

object *Type abbreviation*

Specification: **`type ('l_type)object`**

Description:

Foreign objects have two main components: a *value* part and a *type* part. The value part refers to some raw information contained in an associated store workspace, while the type part defines how that raw information should be interpreted.

The ML type of the language-specific information is provided via the ML type parameter **`'l_type`**.

ReadOnly *Exception*

Specification: **`exception ReadOnly`**

Description:

See **`WriteOnly`**, below.

WriteOnly *Exception*

Specification: **`exception WriteOnly`**

Description:

The **`WriteOnly`** and **`ReadOnly`** exceptions are raised when an object attempts to access or update a store in a manner forbidden by the store's current read / write status. See the datatype **`object_mode`**, below.

object_mode *Datatype*

Specification:

```
datatype object_mode =
    LOCAL_OBJECT | REMOTE_OBJECT
```

Description:

Every object has an associated *mode* which governs the way in which foreign data can be accessed. In general, foreign objects access foreign data that is present locally, in store objects they are associated with.

However, some foreign objects access raw foreign data that is not local to a store but somewhere remote. You may not need to copy the foreign data to a store to do what you want with such data. A local access method based around stores is not appropriate in that case.

Object may therefore be in one of two modes: local or remote. An object in local mode can only access and modify data present within its associated store. An object in remote mode is located remotely to enable it to access foreign data without having first copied it back to a store. In addition, a remote object cannot modify or affect foreign data.

The modes provided are:

LOCAL_OBJECT Foreign data is sited locally within a store workspace. The data can be read and written by ML and foreign code.

Pointer values are not restricted, that is, they can be simple indices (that is, relative values) or machine addresses.

REMOTE_OBJECT Foreign data is sited remotely somewhere in the user's address space. The data can only be read by ML. It cannot be written by ML.

Pointer values are restricted to machine addresses.

objectMode *Function*

Signature

```
val objectMode :
    ('l_type) object -> object_mode
```

Description:

This function returns the current **`object_mode`**.

object_status *Datatype*

Specification:

```
datatype object_status =
    PERMANENT_OBJECT | TEMPORARY_OBJECT
```

Description:

As foreign data is not stored directly as part of a foreign object, objects can be cheaply replicated without changing the meaning of the foreign data. However, it is also sometimes useful to be able to control the way in which foreign objects are replicated.

To do this, each object is given a status value, which can be either *permanent* or *temporary*. The purpose of the object status is that permanent foreign objects can be duplicated but temporary objects are *never* duplicated and would be returned unmodified. Temporary objects are made by an operation that first duplicates a permanent object and changes the status of the duplicate to temporary.

The functions which perform such duplication may need to take suitable care of the language-specific part of an object. As such, these function are provided as part of the language specific interfaces.

The object status values are:

`PERMANENT_OBJECT`

An object with permanent status usually represents some sort of 'live' object which is in some way persistent. By default, newly built objects are given permanent status.

`TEMPORARY_OBJECT`

An object with temporary status usually represents an ephemeral (short-lived) object that is summoned into existence to perform a very specific role in a program.

objectStatus *Function*

Signature:

```
val objectStatus :
    ('l_type) object -> object_status
```

Description:

This function returns the current status of the given object.

OutOfBounds *Exception*

Specification: **`exception OutOfBounds`**

Description:

This exception is raised if an attempt is made to 'move' or 'relocate' an object to some location outside the current store. It is analogous to the **`Subscript`** error that is raised upon an attempt to update an array at an invalid index.

Currency *Exception*

Specification: **`exception Currency`**

Description:

This exception is raised upon an attempt to perform some action upon an object when the association between object and foreign data is assumed to be corrupt or invalid.

The notion of data corruption or validity is naturally dependent upon the interpretation placed on the semantics of the data model of the language being interfaced with. In general, an object is assumed *not* current if it has just been moved, relocated or otherwise changed without its language-dependent interpretation (that is, its 'type') having been adjusted accordingly.

objectCurrency *Function*

Signature: **`val objectCurrency : ('l_type) object -> bool`**

Description:

This predicate reports true if and only if the object supplied is assumed to represent current foreign data.

objectSize *Function*

Signature: `val objectSize : ('l_type) object -> int`

Description:

This function returns the current size, in bytes, of the foreign data located in the store.

objectLocation *Function*

Signature: `val objectLocation : ('l_type) object -> int`

Description:

This function returns the location of the associated foreign data in the store.

objectAddress *Function*

Signature:
```
val objectAddress :
        ('l_type) object -> address
```

Description:

This function returns the machine address of the location of the foreign data in the store.

7.7 The Aliens structure

The **`Aliens`** structure is involved with managing externally linked foreign code objects. This interface uses dynamic linking of foreign code (supplied via the underlying OS) and ML makes a record of what objects have been linked in so far. The basic interface for linking foreign code allows code to either be linked immediately (that is, at link time) or when something actually makes use of the code (that is, at call time).

The following functions are used to ensure that the appropriate associations between ML values representing external objects are in the desired state. These facilities could be used to ‘reset’ associations between foreign code and ML representations and also ensure that up-to-date versions have been obtained.

ensureAliens *Function*

Signature: `val ensureAliens : unit -> unit`

Description:

Ensures that all objects and associated values are available. This preserves any existing entities that are present.

resetAliens *Function*

Signature: `val resetAliens : unit -> unit`

Description:

Reset all objects and associated values, so that they are obtained afresh when they are next requested, and not before. This allows lazy semantics for establishing associations.

refreshAliens *Function*

Signature: `val refreshAliens : unit -> unit`

Description:

Refresh objects and associated values immediately. This re-obtains all external entities, even if they seem to be present.

7.8 The LibML structure

This structure provides the ML side of a C programmers’ interface for accessing ML values and calling ML functions from C. This facility is provided only for C code that has already been called from ML. An application of this is to provide windowing *callback* functions as ML functions. To make use of this, ML values have first to be registered by ML for access from C. The following functions provide facilities for this registration. Values so registered are called *external values*:

registerExternalValue *Function*

Signature: **`val registerExternalValue : string * 'a -> unit`**

Description:

Associates a given value with a string. This string is then used from C as a handle for the C version of the object.

If a value is already associated then an exception is raised.

deleteExternalValue *Function*

Signature: **`val deleteExternalValue : string -> unit`**

Description:

Provides means for deleting a specific external value entry.

externalValues *Function*

Signature: **`val externalValues : unit -> string list`**

Description:

Provides a list of all strings used to name the external values.

clearExternalValues *Function*

Signature: **`val clearExternalValues : unit -> unit`**

Description:

Provides an efficient way to clear all the external value entries in a single operation.

7.9 The Diagnostic structure

The **`Diagnostic`** structure contains a general collection of tools to assist with the provision of diagnostics involving general elements of the FI, such as stores. It is not envisaged that these would be used to provide functionality within applications, but, of course, this is not prohibited either.

Note: These tools may be changed or removed altogether in future versions of the FI.

viewStore *Function*

Signature: **`val viewStore : store -> string`**

Description:

Outputs a string containing information about stores.

dispStore *Function*

Signature: **`val dispStore : store -> store`**

Description:

Outputs the string produced by **`viewStore()`** on the standard output stream and returns the store.

storeInfo *Function*

Signature:

```
val storeInfo : store ->
{ kind : string,
origin : address,
status : string,
alloc : string,
overflow : string,
size : int,
top : int,
free : int }
```

Description:

Provides a structured, diagnostic 'view' of the internals of a store. It can be used to monitor what is happening within a store. It also provides a basis for constructing your own diagnostic tools.

storeData *Function*

Signature: **`val storeData : { store : store, start : int, length : int } -> int list`**

Description:

Provides a region of the store workspace as a list of integers (actually positive numbers from 0 to 255 inclusive).

storeDataHex *Function*

Signature: **`val storeDataHex : { store : store, start : int, length : int } -> string`**

Description:

Provides a region of the store workspace as a hex string.

storeDataAscii *Function*

Signature: **`storeDataAscii : { store : store, start : int, length : int } -> string`**

Description:

Provides a region of the store workspace as an ASCII string.

diffAddr *Function*

Signature: **`val diffAddr : address -> address -> int`**

Description:

Yields the difference of two addresses. Useful for relative address computations.

incrAddr *Function*

Signature: **val incrAddr : address * int -> address**

Description:

Offsets a given address by an integer. Note that the offset may be positive or negative.

7.10 The C structure

This structure contains the language-specific part of the interface providing support for C. It has the following sub-structures:

```
structure Structure
structure Type
structure Value
structure Signature
structure Function
structure Diagnostic
```

The basic idea behind this part of the FI is to provide support for a C-compatible data model within ML. Foreign data representation is provided under this model via the **object** type.

Objects may be considered to have two main components: a *value* part and a *type* part. The value part of an object consists of the raw information concerning what is being denoted, whereas the type part specifies how the value part can be interpreted. Both components of objects are under the control of the ML programmer and can be manipulated in ways that are familiar to a C programmer. Furthermore, the value parts of objects are associated with physical storage via a given store workspace, thereby ensuring that memory allocation is (i) static and (ii) decoupled from the representation of specific data values.

This structure also provides an interface for robustly managing dynamically linked foreign code and invoking it. This is provided via the **C.Structure** and **C.Signature** sub-structures. The basic idea here is that linked-in foreign code provides raw behavior, which is kept locked within a **c_structure** object. To access and use this raw behavior, another object called a **c_signature** is needed to provide signature information as **c_type** values. When appropriately matching **c_signature** and **c_structure** objects are combined, this permits the raw behavior contained within the **c_structure** to be invoked.

7.10.1 The C.Structure structure

This structure provides facilities for loading and dynamically linking foreign code for use with the C data model.

c_structure *Type abbreviation*

Specification: **`type c_structure`**

Description:

Objects of type **`c_structure`** are containers of foreign code. Each **`c_structure`** object is created as a result of dynamically linking foreign code from a file.

No attempt is made at present to cache this code when images are saved, and so it would have to be restored when an image is restarted. Fortunately, this is made trivial by using the tools provided via the **`Aliens`** structure. See “The Aliens structure” on page 231.

load_mode *Datatype*

Specification: **`datatype load_mode = IMMEDIATE_LOAD | DEFERRED_LOAD`**

Description:

When foreign code is loaded, the dynamic linking of that code may occur immediately (at load time) or later (at call time). These options are reflected here by:

`IMMEDIATE_LOAD` Link foreign code immediately.

`DEFERRED_LOAD` Link foreign code at first call to the library.

loadObjectFile *Function*

Signature: **`val loadObjectFile : filename * load_mode -> c_structure`**

Description:

This function generates a **c_structure** by dynamically linking foreign code associated with the specified file, in accordance with the given **load_mode**.

fileInfo *Function*

Signature: **val fileInfo : c_structure -> (filename * load_mode)**

Description:

This function obtains both the filename and the **load_mode** used to create the **c_structure**.

filesLoaded *Function*

Signature: **val filesLoaded : unit -> filename list**

Description:

This yields a list of all foreign code files loaded so far.

symbols *Function*

Signature: **val symbols : c_structure -> name list**

Description:

Extracts symbol table information concerning the foreign code contained within a given **c_structure**. This info might indicate the name of the object, what kind of object it is, and even a relocatable address associated with the code.

value_type *Datatype*

Specification: **datatype value_type = CODE_VALUE | VAR_VALUE | UNKNOWN_VALUE**

Description:

This datatype provides a coarse classification of foreign code objects.

CODE_VALUE Object appears to be functional code of some description.

VAR_VALUE Object appears to be a (visible) variable containing foreign data.

UNKNOWN_VALUE Object cannot be classified, though it could be either of the above.

symbolInfo *Function*

Signature: **val symbolInfo : c_structure * name -> value_type**

Description:

This function attempts to classify named foreign code objects according to the scheme given in **value_type**, above.

7.10.2 The C.Type structure

This structure provides support for representing C type information in a manipulable form as ML data.

enum_value *Type abbreviation*

Specification: **type enum_value**

Description:

This type is used to model enumerated values. There are conversion functions to and from integers. Equivalent to **string**.

tag *Type abbreviation*

Specification: **type tag**

Description:

This is used to provide convenient ‘type’ tags. Equivalent to **string**.

pointer_kind *Datatype*

Specification: **`datatype pointer_kind = LOCAL_PTR | RELATIVE_PTR | REMOTE_PTR`**

Description:

As described in earlier sections, machine pointers need to be treated with special care. To provide this care, the idea of a ‘pointer kind’ is introduced. This provides qualification of pointer values and determines how they can be used. A pointer kind is one of the following:

`LOCAL_PTR` Machine address pointing within the associated store.

`REMOTE_PTR` Machine address pointing within user-accessible memory.

`RELATIVE_PTR` Index value accessing location within associated store.

c_type *Datatype*

Specification:

```
datatype c_type =
    VOID_TYPE | CHAR_TYPE | ...
```

Description:

The ML type **`c_type`** provides a representation of C type information accessible as an ML value. These values are used to provide information on how to interpret the value parts of **`c_object`** objects used to represent foreign data.

`VOID_TYPE` This represents the C type **`void`**. Its size is zero bytes.

`CHAR_TYPE` This represents the C type **`char`**. Its size is 1 byte. It may be associated with either signed or unsigned chars by the particular C compiler used.

`UNSIGNED_CHAR_TYPE`

This represents the C type **`unsigned char`**.

`SIGNED_CHAR_TYPE`

This represents the C type **`signed char`**.

SHORT_TYPE This represents the C type **short int**.

INT_TYPE This represents the C type **int**.

LONG_TYPE This represents the C type **long int**.

UNSIGNED_SHORT_TYPE

This represents the C type **unsigned short int**.

UNSIGNED_INT_TYPE

This represents the C type **unsigned int**.

UNSIGNED_LONG_TYPE

This represents the C type **unsigned long int**.

FLOAT_TYPE This represents the C type **float**.

DOUBLE_TYPE This represents the C type **double**.

LONG_DOUBLE_TYPE

This represents the C type **long double**.

STRING_TYPE of { length : int }

This represents C string type **char*** where each string has an explicit amount of storage allocated for it. This length should include room for the null byte sentinel.

TYPENAME of { name : name, defn : c_type, size : int }

This represents the use of a named type (that is, a name created with **typedef**) within a C type.

POINTER_TYPE of { ctype : c_type, mode : pointer_kind ref }

This represents ANSI C's idea of typed pointers. The additional pointer mode information concerns how ML encodes and treats this pointer information. In particular, 'relative' pointers are simply small indices which make indirection within a store workspace more direct and efficient. The representation also caters for 'remote' pointers which can refer to arbitrary places in memory, and 'local' pointers are remote pointers which are known to refer to places in the associated store workspace.

`STRUCT_TYPE of { tag : tag option, fields : c_field list, size : int }`

This represents the structured record type in C. Such types may be tagged or untagged.

`UNION_TYPE of { tag : tag option, variants : c_variant list ref, size : int, current : c_variant }`

This represents the union type in C. Such types may be tagged or untagged.

`ARRAY_TYPE of { length : int, ctype : c_type }`

This represents the array type in C.

`ENUM_TYPE of { tag : tag option, elems : enum_value list, card : int }`

This represents (simple) enumerated types as provided by C. Such types may be tagged or untagged.

c_variant *Datatype*

Specification: **`datatype c_variant = VARIANT of { name : name, ctype : c_type, size : int }`**

Description:

Used to encode members of C unions. Note that each variant object contains its size.

c_field *Datatype*

Specification: **`datatype c_field = FIELD of { name : name, ctype : c_type, size : int, padding : int, offset : int }`**

Description:

This is used to encode field components of C structs. In addition to its size, this representation also has the offset for the field from the start of the record (useful for indexing) and also takes account of any ‘padding’ required within each field. This padding depends upon the particular compiler used to compile foreign code.

sizeOf *Function*

Signature: **`val sizeOf : c_type -> int`**

Description:

This function computes the size of a **`c_type`** object and fills in any size attributes that have not already been computed. Clearly, this requires named types to have had declarations filled in — with failure if they are not.

equalType *Function*

Signature: **`val equalType : c_type * c_type -> bool`**

Description:

Because **`c_type`** values can contain size attribute (which may be set to **`NONE`**), this function is used to make equality comparisons between two **`c_types`** which disregard the attribute components they may possess.

Some convenience functions for building compound **`c_type`** objects:

structType *Function*

Signature: **`val structType : string * (string * c_type) list -> c_type`**

Description:

Builds C struct type representations. Note that these C type objects are tagged.

unionType *Function*

Signature: **`val unionType : string * (string * c_type) list -> c_type`**

Description:

Builds C union type representations. Note that these C type objects are tagged.

ptrType *Function*

Signature: **`val ptrType : c_type -> c_type`**

Description:

Builds C pointer type representations.

typeName *Function*

Signature: **`val typeName : string -> c_type`**

Description:

Builds C named type objects.

enumType *Function*

Signature: **`val enumType : string * string list -> c_type`**

Description:

Builds C enumerated type representations

7.10.3 The C.Value structure

This structure provides support for foreign data values as ML data structures.

store

Type abbreviation

Specification: **`type store`**

Description:

Equivalent to **`Store.store`**. See page 224.

object_mode

Type abbreviation

Specification: **`type object_mode`**

Description:

Equivalent to **`Object.object_mode`**. See page 228.

c_type

Type abbreviation

Specification: **`type c_type`**

Description:

Equivalent to **`Type.c_type`**. See page 240.

c_object

Type abbreviation

Specification: **`type c_object`**

Description:

This is an encapsulated ML type used to represent foreign data and is equivalent to **`(c_type)Object.object`**. See page 227.

object

Function

Signature:

```
val object : { ctype : c_type, store : store }
-> c_object
```

Description:

This generates fresh **`c_objects`**, given specific type information and a particular store to contain the raw value information.

setObjectMode *Function*

Signature: **`val setObjectMode : c_object * object_mode -> unit`**

Description:

This is used to change the current object mode.

objectType *Function*

Signature: **`val objectType : c_object -> c_type`**

Description:

This is used to inspect the current **`c_type`**.

castObjectType *Function*

Signature: **`val castObjectType : c_object * c_type -> unit`**

Description:

This is used to change the current **`c_type`**.

tmpObject *Function*

Signature: **`val tmpObject : c_object -> c_object`**

Description:

This maps permanent objects into a duplicate except that the status of the duplicate is mapped to temporary. Temporary objects are simply returned.

dupObject *Function*

Signature: **`val dupObject : c_object -> c_object`**

Description:

Duplicates permanent objects, but does not duplicate objects whose status is temporary.

newObject *Function*

Signature: **`val newObject : c_object -> c_object`**

Description:

This generates a fresh foreign object, including making a duplicate type component (using **`dup_type()`**), irrespective of the object's status.

c_char *Type abbreviation*

Specification: **`type c_char`**

Description:

This ML type is compatible with the C type **`char`**.

c_short_int *Type abbreviation*

Specification: **`type c_short_int`**

Description:

This ML type is compatible with the C type **`short int`**.

c_int *Type abbreviation*

Specification: **`type c_int`**

Description:

This ML type is compatible with the C type **`int`**.

c_long_int *Type abbreviation*

Specification: type c_long_int

Description:

This ML type should be compatible with the C type **`long int`**.

c_real
Type abbreviation

Specification: `type c_real`

Description:

This ML type is compatible with the C type **float**.

c_double
Type abbreviation

Specification: `type c_double`

Description:

This ML type is compatible with the C type **double**.

c_long_status
Type abbreviation

Specification: `type c_long_double`

Description:

This ML type should be compatible with the C type **long double**.

ForeignType
Exception

StoreAccess
Exception

OutOfBounds
Exception

Currency
Exception

The following are generally ‘setter’ functions for particular kinds of C data. In particular, they expect the foreign objects to have an appropriate **c_type** already set. If not, then they fail with exception **ForeignType**.

setChar
Function

Signature: `val setChar : c_object * c_char -> unit`

Description:

This function sets the object value to a value representing a C **char**.

setUnsignedChar *Function*

Signature: **`val setUnsignedChar : c_object * c_char -> unit`**

Description:

This function sets the object value to a value representing an unsigned C char (that is, 0 <= char <= 255).

setSignedChar *Function*

Signature: **`val setSignedChar : c_object * c_char -> unit`**

Description:

This function sets the object value to a value representing an unsigned C character (that is -127 <= **char** <= 128).

setShort *Function*

Signature: **`val setShort : c_object * c_short_int -> unit`**

Description:

This function sets the object value to a value representing a C short integer.

setInt *Function*

Signature: **`val setInt : c_object * c_int -> unit`**

Description:

This function sets the object value to a value representing a C integer.

setLong *Function*

Signature: **`val setLong : c_object * c_long_int -> unit`**

Description:

This function sets the object value to a value representing a C long integer.

setUnsignedShort *Function*

Signature: **`val setUnsignedShort : c_object * c_short_int -> unit`**

Description:

This function sets the object value to a value representing a C unsigned short integer.

setUnsigned *Function*

Signature: **`val setUnsigned : c_object * c_int -> unit`**

Description:

This function sets the object value to a value representing a C unsigned integer.

setUnsignedLong *Function*

Signature: **`val setUnsignedLong : c_object * c_long_int -> unit`**

Description:

This function sets the object value to a value representing a C unsigned long integer.

setWord32 *Function*

Signature: **`val setWord32 : c_object * word32 -> unit`**

Description:

This function sets the object value from an ML 32-bit value.

setFloat

Function

Signature: **`val setFloat : c_object * c_real -> unit`**

Description:

This function sets the object value to a value representing a C floating-point real value.

setDouble

Function

Signature: **`val setDouble : c_object * c_double -> unit`**

Description:

This function sets the object value to a value representing a C double floating point real value.

setLongDouble

Function

Signature: **`val setLongDouble : c_object * c_long_double -> unit`**

Description:

This function sets the object value to a value representing a C long double floating-point real value.

setString

Function

Signature: **`val setString : c_object * string -> unit`**

Description:

This function sets the object value to a value representing a C string. In general, ML strings can contain embedded NULL characters — so only the string up to the first NULL is significant. However, if no NULL is included then one is added. Finally, the foreign object must have a suitable string **`c_type`** whose length (including any NULL sentinel) is sufficient to contain this data.

Functions for manipulating pointer objects:

setAddr

Function

Signature: **`val setAddr : { obj:c_object, addr:c_object } -> unit`**

Description:

This makes the value part of the **`obj`** object coincide with the value based at the address given by the **`addr`** object. The **`c_type`** of **`obj`** may be arbitrary and the **`c_type`** of **`addr`** should be a numeric type capable of representing a machine address or an appropriate pointer type.

In a sense, this makes the **`obj`** object inspect value data at a given address.

setPtrAddr

Function

Signature: **`val setPtrAddr : { ptr:c_object, addr:c_object } -> unit`**

Description:

This sets the given pointer object **`ptr`** to reference the address value given by **`addr`** (see above). The **`c_type`** of **`ptr`** is any pointer **`c_type`** and the **`c_type`** of **`addr`** is any numeric type capable of representing a machine address or an appropriate pointer type.

This function makes a pointer object refer to a given address.

setPtrAddrOf

Function

Signature: **`val setPtrAddrOf : { ptr:c_object, data:c_object } -> unit`**

Description:

This sets the given pointer object, **`ptr`**, to reference the value referred to by the **`data`** object. The **`c_type`** of **`ptr`** is any pointer **`c_type`** and the **`c_type`** of **`data`** must be compatible with this.

This function makes a pointer object refer to a given piece of data of compatible type.

setPtrData *Function*

Signature: **`val setPtrData : { ptr:c_object, data:c_object } -> unit`**

Description:

This sets the data that is addressed by the pointer object, **`ptr`**, to the data specified by the object **`data`**. The **`c_type`** of **`ptr`** can be any pointer **`c_type`** and the **`c_type`** of **`data`** must be compatible with this.

This function indirectly assigns data into the space referred to by pointer.

setPtrType *Function*

Signature: val setPtrType : { ptr:c_object, data:c_object } -> unit

Description:

This sets the **`c_type`** of the data addressed by the pointer **`ptr`** to the **`c_type`** specified by the object **`data`**. The **`c_type`** of **`ptr`** can be any pointer **`c_type`** and the **`c_type`** of **`data`** can be arbitrary. The current pointer mode is preserved.

This function performs an implicit type cast of the pointer to match that of the given data object.

castPtrType *Function*

Signature: **`val castPtrType : { ptr : c_object, ctype : c_type } -> unit`**

Description:

This sets the **`c_type`** of the data addressed by the pointer **`ptr`** to the **`c_type`** specified. This function performs an explicit type cast of the given pointer, while preserving the current pointer mode.

setLocalPtr *Function*

Signature: **`val setLocalPtr : c_object -> unit`**

Description:

This converts the current pointer into a local pointer — that is, a machine address located within the associated store workspace. This may fail if the given pointer is a remote pointer that points outside of this workspace.

setRelativePtr *Function*

Signature: **`val setRelativePtr : c_object -> unit`**

Description:

This converts the current pointer into a relative pointer — that is, a small index value giving the relative offset from the origin address of the store workspace. This fails if the given pointer points outside the associated store workspace.

setRemotePtr *Function*

Signature: **`val setRemotePtr : c_object -> unit`**

Description:

This converts the current pointer into a remote pointer — that is, a machine address.

Functions for manipulating structured objects:

setStruct *Function*

Signature: **`val setStruct : c_object * (c_object list) -> unit`**

Description:

This function takes an object specifying a structure and updates its fields from the given list of data items. This relies upon fields being ordered in a structure and that the **`c_types`** of corresponding items and fields are matched. If there are fewer items than fields then only the corresponding leading prefix of fields are updated. Also, if there are more items than fields then the excess items are ignored.

setField

Function

Signature: **`val setField : { record : c_object, field:name, data : c_object } -> unit`**

Description:

This function updates a specific field of a C struct with the given data.

setMember

Function

Signature: **`val setMember : { union : c_object, member : name } -> unit`**

Description:

This updates an object with union **`c_type`** by selecting a particular member. The selected member must be one of the known options.

setUnion

Function

Signature: **`val setUnion : { union : c_object, data : c_object } -> unit`**

Description:

This updates an object with union **`c_type`** with given data. The **`c_type`** of the current member of the union object must be compatible with the **`c_type`** of the data.

setArray

Function

Signature: **`val setArray : c_object * (c_object list) * int -> unit`**

Description:

This updates an array object with a ‘slice’ of items, based at a given index. This allows several elements of an array to be updated together. The array elements updated begin with the given index and continue with consecutive indices until either the list is exhausted or the array ends.

setEnum *Function*

Signature: **`val setEnum : c_object * int -> unit`**

Description:

This updates an object containing enumerated values. The integer must be in the appropriate range defined by the **`c_type`** of the object. The **`c_type`** of the object should be an enumerated type.

The following are particular selection functions for particular kinds of structured C data — they expect the foreign objects to have an appropriate **`c_type`** already set. If not, they fail with exception **`ForeignType`**.

indexObject *Function*

Signature: **`val indexObject : { array:c_object, tgt:c_object, index:int } -> unit`**

Description:

This selects an array element from the given array at the given **`index`** and copies the data to the target object, **`tgt`**. The **`index`** must be in the range of the array; the **`c_type`** of **`array`** should be an array type; and the target object should have compatible **`c_type`**.

derefObject *Function*

Signature: **`val derefObject : { ptr:c_object, tgt:c_object } -> unit`**

Description:

This locates the data pointed at by the pointer object and copies it to the target object.

selectObject *Function*

Signature: **`val selectObject : { record:c_object, tgt:c_object, field:name } -> unit`**

Description:

This selects data from a field of a given record and copies it to the target object. The field has to be one of those associated with the C struct type of the **record**; the **c_type** of the target object must also be compatible with the field.

coerceObject *Function*

Signature: **val coerceObject : { union:c_object, tgt:c_object } -> unit**

Description:

This extracts the content of the union object and copies it to the target object. The **c_types** of the union and the target do not have to match (that is, implicit coercion).

copyIndexObject *Function*

Signature: **val copyIndexObject : c_object * int -> c_object**

Description:

As **indexObject** (page 256), but generates a new object to provide the result.

copyDerefObject *Function*

Signature: **val copyDerefObject : c_object -> c_object**

Description:

As **derefObject** (page 256), but generates a new object to provide the result.

copySelectObject *Function*

Signature: **val copySelectObject : c_object * name -> c_object**

Description:

As **selectObject** (page 256), but generates a new object to provide the result.

copyCoerceObject *Function*

Signature: **val copyCoerceObject : c_object -> c_object**

Description:

As for **coerceObject()** above, but generates a new object to provide the result.

indexObjectType *Function*

Signature: **val indexObjectType : c_object -> c_type**

Description:

This gives the **c_type** of an element of the array.

derefObjectType *Function*

Signature: **val derefObjectType : c_object -> c_type**

Description:

This gives the **c_type** of the value pointed at by the pointer object.

selectObjectType *Function*

Signature: **val selectObjectType : c_object * name -> c_type**

Description:

This gives the **c_type** of the field selected from the C struct object.

coerceObjectType *Function*

Signature: **val coerceObjectType : c_object -> c_type**

Description:

This gives the **c_type** of the current member of the C union object.

indexObjectSize *Function*

Signature: `val indexObjectSize : c_object -> int`

Description:

This gives the size (in bytes) of an element of the specified array.

derefObjectSize *Function*

Signature: `val derefObjectSize : c_object -> int`

Description:

This gives the size (in bytes) of the value pointed at by the pointer object.

selectObjectSize *Function*

Signature: `val selectObjectSize : c_object * name -> int`

Description:

This gives the size (in bytes) of the field selected from the C struct object.

coerceObjectSize *Function*

Signature: `val coerceObjectSize : c_object -> int`

Description:

This gives the size (in bytes) of the current member for the C union object.

nextArrayItem *Function*

Signature: `val nextArrayItem : c_object -> unit`

Description:

This shifts the object forwards through the workspace by an amount equal to its size. This is useful when stepping through an array.

prevArrayItem *Function*

Signature: **`val prevArrayItem : c_object -> unit`**

Description:

As for **`nextArrayItem`**, except that the object is shifted backwards.

The following are generally 'getter' functions for particular kinds of C data — as for the related setter functions, they expect the **`object`** objects to have an appropriate **`c_type`** already set. If not, then they fail with exception **`ForeignType`**.

getChar *Function*

Signature: **`val getChar : c_object -> c_char`**

Description:

Extracts a C character (represented as an ML value) from an object with appropriate **`c_type`**.

getUnsignedChar *Function*

Signature: **`val getUnsignedChar : c_object -> c_char`**

Description:

Extracts a C unsigned character (represented as an ML value in the range 0–255) from an object with appropriate **`c_type`**.

getSignedChar *Function*

Signature: **`val getSignedChar : c_object -> c_char`**

Description:

Extracts a C signed character represented as an ML value (-127–127) from an object with appropriate **`c_type`**.

getShort *Function*

Signature: **`val getShort : c_object -> c_short_int`**

Description:

Extracts a C short value (represented as an ML value) from an object with appropriate **`c_type`**.

getInt *Function*

Signature: **`val getInt : c_object -> c_int`**

Description:

Extracts a C int value (represented as an ML value) from an object with appropriate **`c_type`**.

getLong *Function*

Signature: **`val getLong : c_object -> c_long_int`**

Description:

Extracts a C long value (represented as an ML value) from an object with appropriate **`c_type`**.

getUnsignedShort *Function*

Signature: **`val getUnsignedShort : c_object -> c_short_int`**

Description:

Extracts a C unsigned short value (represented as an ML value) from an object with appropriate **`c_type`**.

getUnsigned *Function*

Signature: **`val getUnsigned : c_object -> c_int`**

Description:

Extracts a C unsigned int value (represented as an ML value) from an object with appropriate **`c_type`**.

getUnsignedLong *Function*

Signature: **`val getUnsignedLong : c_object -> c_long_int`**

Description:

Extracts a C unsigned long value (represented as an ML value) from an object with appropriate **`c_type`**.

getWord32 *Function*

Signature: **`val getWord32 : c_object -> word32`**

Description:

Extracts a 4-byte quantity (that is, C unsigned) from an object with appropriate **`c_type`**. The quantity is represented as a **`Word32`** ML value.

getFloat *Function*

Signature: **`val getFloat : c_object -> c_real`**

Description:

Extracts a C float (represented as an ML value) from an object with appropriate **`c_type`**.

getDouble *Function*

Signature: **`val getDouble : c_object -> c_double`**

Description:

Extracts a C double float (represented as an ML value) from an object with appropriate **`c_type`**.

getLongDouble *Function*

Signature: **`val getLongDouble : c_object -> c_long_double`**

Description:

Extracts a C long double float (represented as an ML value) from an object with appropriate **c_type**.

getString *Function*

Signature: **`val getString : c_object -> string`**

Description:

Extracts an ASCII character string (represented as an ML value) from an object with appropriate **c_type**.

getData *Function*

Signature: **`val getData : c_object -> c_object`**

Description:

Yields an object containing the dereferenced value of the given pointer. This is a synonym for the **copyDerefObject** function. See page 257.

getStruct *Function*

Signature: **`val getStruct : c_object -> c_object list`**

Description:

Yields a list of objects each corresponding to a field of the given C struct object.

getField *Function*

Signature: **`val getField : c_object * name -> c_object`**

Description:

Yields the value of a given field. This is a synonym for the **copySelectObject** function. See page 257.

getUnion *Function*

Signature: **`val getUnion : c_object -> c_object`**

Description:

Yields an object whose value and **c_type** correspond to the current member of the given union object. This is a synonym for the **copyCoerceObject** function. See page 258.

getArray *Function*

Signature: **val getArray : c_object -> c_object list**

Description:

Yields a list of objects corresponding to the elements of the given array object.

getEnum *Function*

Signature: **val getEnum : c_object -> int**

Description:

Yields an integer corresponding to the enumerated value represented by the object given.

7.10.4 The C.Signature structure

This structure defines how external signature information is represented and provides operators for manipulating this information.

c_type *Type abbreviation*

Specification: **type c_type**

Description:

Representation for C-type information. Equivalent to **Type.c_type**.

c_signature *Type abbreviation*

Specification: **type c_signature**

Description:

This is an encapsulated abstract type for representing consistent collections of C declaration information for types, functions and variables.

c_decl

Datatype

Specification:

```
datatype c_decl = UNDEF_DECL | VAR_DECL of {
name : name, ctype : c_type } | FUN_DECL of {
name   : name, source : c_type list, target :
c_type } | TYPE_DECL of { name : name, defn :
c_type, size : int } | CONST_DECL of { name :
name, ctype : c_type }
```

Description:

This data type is used to represent C type declaration info and has the following structure:

`UNDEF_DECL` This value is included as a default return value for queries rather than using option values (that is, **`NONE`** and **`SOME`**) for wrapping and unwrapping these values.

`VAR_DECL of { name : name, ctype : c_type }`

C variable declarations can be recorded in this form. The type information may be updated and modified.

`FUN_DECL of { name : name, source : c_type list, target : c_type }`

C function declaration information can be recorded in this form. The type information may be updated and modified.

`TYPE_DECL of { name : name, defn : c_type, size : int }`

C type information associated with a name (that is, **`typedef`** and **`struct`** / **`union`** / **`enum`** declarations) can be recorded in this form. The associated type and size information may be updated and modified.

CONST_DECL of { name : name, ctype : c_type }

Type information associated with simple literal **#define** constants can be recorded in this form.

newSignature *Function*

Signature: **val newSignature : unit -> c_signature**

Description:

This generates a fresh **c_signature** object.

lookupEntry *Function*

Signature: **val lookupEntry : c_signature -> name -> c_decl**

Description:

This takes a **c_signature** and a name and returns a declaration value having that name, if one exists.

defEntry *Function*

Signature: **val defEntry : c_signature * c_decl -> unit**

Description:

This updates a **c_signature** object by adding a given entry.

removeEntry *Function*

Signature: **val removeEntry : c_signature * name -> unit**

Description:

This removes the named entry from the given **c_signature** object.

showEntries *Function*

Signature: **val showEntries : c_signature -> c_decl list**

Description:

This yields a list of all the entries contained within a given **`c_signature`**.

normaliseType *Function*

Signature: **`val normaliseType : c_signature -> (c_type -> c_type)`**

Description:

This function takes a **`c_type`** object and ensures that size information is correct and up to date. Normalised types can have their sizes computed using **`sizeOf`**. See page 243.

7.10.5 The C.Function structure

c_structure *Type abbreviation*

Specification: **`type c_structure`**

Description:

Equivalent to **`Structure.c_structure`**.

c_signature *Type abbreviation*

Specification: **`type c_signature`**

Description:

Equivalent to **`Signature.c_signature`**.

c_type *Type abbreviation*

Specification: **`type c_type`**

Description:

Equivalent to **`Type.c_type`**.

c_object

Type abbreviation

Specification: **`type c_object`**

Description:

Equivalent to **`Value.c_object`**.

c_function

Type abbreviation

Specification: **`type c_function`**

Description:

This is an encapsulated abstract type used for representing foreign function data. It supports sufficient information to enable these functions to be called with appropriate arguments and for its results to be interpreted.

defineForeignFun

Function

Signature:

```
val defineForeignFun : (c_structure *
c_signature) -> (name -> c_function)
```

Description:

This is the main function in which all the key elements of the C interface are combined. This function is used to extract named foreign code from a **`c_structure`** and then combined with the type information associated with the **`c_signature`** for that name. The result is a **`c_function`** object which can then be supplied with arguments and called.

call

Function

Signature:

```
val call : c_function -> (c_object list *
c_object) -> unit
```

Description:

This function takes a **`c_function`** object and a list of objects representing the arguments, calls the associated foreign function and returns the result to the other given object. Of course, all the type information for **`c_function`**, argument objects and result object must match accordingly.

7.10.6 The C.Diagnostic structure

This structure contains a general collection of tools to help provide diagnostic services for C specific parts of the FI such as **`c_objects`**. It is not envisaged that these would be used to provide functionality within applications, but this is of course not prohibited.

These tools are provided here on the understanding that this part of the interface may be changed arbitrarily, In particular, there is no guarantee to preserve any functionality in future versions. However, such interfaces are not changed without there being just cause.

store *Type abbreviation*

Specification: **`type store`**

Description:

Equivalent to **`Store.store`**.

c_type *Type abbreviation*

Specification: **`type c_type`**

Description:

Equivalent to **`Type.c_type`**.

c_object *Type abbreviation*

Specification: **`type c_object`**

Description:

Equivalent to **`Value.c_object`**.

cTypeInfo *Function*

Signature: **`val cTypeInfo : c_type -> string`**

Description:

This provides a string describing the given **`c_type`** value.

viewObject *Function*

Signature: **`val viewObject : c_object -> string`**

Description:

This provides a string describing the given **`c_object`**.

dispObject *Function*

Signature: **`val dispObject : c_object -> c_object`**

Description:

Outputs the string produced by **`viewObject`**, above, on the standard output stream and returns the **`c_object`**.

objectInfo *Function*

Signature: **`val objectInfo : c_object -> { store : store, status : string, currency : string, mode : string, langtype : string, size : int, base : address option, offset : int }`**

Description:

This provides a structured, diagnostic view of the internals of a **`c_object`**. This can be used by programmers to construct additional diagnostic tools.

objectData *Function*

Signature: **`val objectData : c_object -> int list`**

Description:

This function presents the data associated with an object in the form of a list of integers.

objectDataHex *Function*

Signature: **`val objectDataHex : c_object -> string`**

Description:

This function presents the data associated with an object in the form of string of hexadecimal digits.

objectDataAscii *Function*

Signature: **`val objectDataAscii : c_object -> string`**

Description:

This function presents the data associated with an object in the form of an ASCII string.

Index

K

L

M

N

X

Y