The Haskell 98 Foreign Function Interface 1.0
An Addendum to the Haskell 98 Report

Manuel Chakravarty [editor], University of New South Wales
Sigbjorn Finne, Galois Connections, Inc.
Fergus Henderson, University of Melbourne
Marcin Kowalczyk, Warsaw University
Daan Leijen, University of Utrecht
Simon Marlow, Microsoft Research, Cambridge
Erik Meijer, Microsoft Corporation
Sven Panne, BetaResearch GmbH
Simon Peyton Jones, Microsoft Research, Cambridge
Alastair Reid, Reid Consulting (UK) Ltd.
Malcolm Wallace, University of York
Michael Weber, University of Aachen

The master version of the Haskell FFI Report is at `haskell.org`. Any corrections or changes in the report are found there.

Contents

1 Introduction 1
- 1.1 Embedding Into Haskell 98 1
- 1.2 Language-Specific FFI Support 1
- 1.3 Contexts 1
- 1.4 Cross Language Type Consistency 2

2 Lexical Structure 3

3 Foreign Declarations 4
- 3.1 Calling Conventions 4
- 3.2 Foreign Types 5
- 3.3 Import Declarations 5
- 3.4 Export Declarations 6

4 Specification of External Entities 6
- 4.1 Standard C Calls 7
  - 4.1.1 Import Declarations 7
  - 4.1.2 Export Declarations 8
  - 4.1.3 Constraints on Foreign Function Types 8
  - 4.1.4 Specification of Header Files 9
  - 4.1.5 C Argument Promotion 9
- 4.2 Win32 API Calls 10

5 Marshalling 11
- 5.1 `Foreign` 11
- 5.2 `Bits` 12
- 5.3 `Int` and `Word` 13
- 5.4 `Ptr` 13
  - 5.4.1 Data Pointers 13
  - 5.4.2 Function Pointers 14
- 5.5 `ForeignPtr` 14
- 5.6 `StablePtr` 16
- 5.7 `Storable` 17
- 5.8 `MarshalAlloc` 18
- 5.9 `MarshalArray` 19
- 5.10 `MarshalError` 20
  - 5.10.1 I/O Errors 20
  - 5.10.2 Result Value Checks 21
- 5.11 `MarshalUtils` 22

6 C-Specific Marshalling 23
- 6.1 `CForeign` 23
- 6.2 `CTypes` 25
- 6.3 `CString` 27
- 6.4 `CError` 28

Preface

The definition of Haskell 98 [7], while being comprehensive with respect to the functional core language, does lack a range of features of more operational flavour, such as a foreign language interface, concurrency support, and fully fledged exception handling. As these features are of central importance to many real world applications of the language, there is a danger that different implementations become de facto incompatible for such applications due to system-specific extensions of the core language. The present FFI specification is aimed at reducing this risk by defining a simple, yet comprehensive extension to Haskell 98 for the purpose of interfacing to program components implemented in a language other than Haskell.

The goal behind this foreign function interface (FFI) specification is twofold: It enables (1) to describe in Haskell the interface to foreign functionality and (2) to use from foreign code Haskell routines. More precisely, its aim is to support the implementation of programs in a mixture of Haskell and other languages such that the source code is portable across different implementations of Haskell and non-Haskell systems as well as independent of the architecture and operating system.

The design as presented in this report builds on experiences with a number of foreign function interfaces that, over time, have been provided by the major Haskell implementations. Central in the final design was the goal to be comprehensive while being simple and minimising changes with respect to Haskell 98; the latter includes to avoid pollution of the name space with new keywords. Consequently, as much as possible of the FFI functionality is realised in the form of libraries. Simplicity generally overruled maximum convenience for the programmer as a design goal. Thus, support for more convenient interface specifications is the domain of system-independent tools that generate code following the present specification.

Acknowledgements

We heartily thank the kind people who assisted us with their comments and suggestions on the `ffi@haskell.org` and `haskell@haskell.org` mailing lists as well as all the users of previous versions of the FFI who helped to shape the development by their feedback. We thank Olaf Chitil, Peter Gammie, Wolfram Kahl, Martin D. Kealey, Ian Lynagh, John Meacham, Ross Paterson, George Russell, and Wolfgang Thaller for errata and additions to previous versions of this report.

1 Introduction

The extension of Haskell 98 defined in this report facilitates the use of non-Haskell code from Haskell and vice versa in a portable manner. Intrusion into Haskell 98 has been kept to a minimum and the defined facilities have been extensively tested with large libraries.

The present Version 1.0 of the FFI report does only fully specify the interaction between Haskell code with code that follows the C calling convention. However, the design of the FFI is such that it enables the modular extension of the present definition to include the calling conventions of other programming languages, such as C++ and Java. A precise definition of the support for those languages is expected to be included in later versions of this report. The second major omission from the present report is the definition of the interaction with multithreading in the foreign language and, in particular, the treatment of thread-local state. Work on this problem is not sufficiently mature to be included into Version 1.0 of the report.

1.1 Embedding Into Haskell 98

The present report is to be regarded as an addendum to the Haskell 98 Report [7]. As such, syntactic and semantic definitions refer to names and definitions in the Haskell 98 Report where appropriate without further explanation. Care has been taken to invalidate as few as possible legal Haskell 98 programs in the process of adding FFI support. In particular, only a single addition to the set of reserved identifiers, namely `foreign`, has been made.

Moreover, it is expected that the present FFI specification will be considered for inclusion into future revisions of the Haskell standard.

1.2 Language-Specific FFI Support

The core of the present specification is independent of the foreign language that is used in conjunction with Haskell. However, there are two areas where FFI specifications must become language specific: (1) the specification of external names and (2) the marshalling of the basic types of a foreign language. As an example of the former, consider that in C [4] a simple identifier is sufficient to identify an object, while Java [2], in general, requires a qualified name in conjunction with argument and result types to resolve possible overloading. Regarding the second point, consider that many languages do not specify the exact representation of some basic types. For example the type `int` in C may be 16, 32, or 64 bits wide. Similarly, the Haskell report guarantees only that `Int` covers at least the range $[-2^{29}, 2^{29}-1]$. As a consequence, to reliably represent values of C's `int` in Haskell, we have to introduce a new type `CInt`, which is guaranteed to match the representation of `int`.

The specification of external names, dependent on a calling convention, is described in Section 4, whereas the marshalling of the basic types in dependence on a foreign language is described in Section 5.

1.3 Contexts

For a given Haskell system, we define the *Haskell context* to be the execution context of the abstract machine on which the Haskell system is based. This includes the heap, stacks, and the registers of the abstract machine and their mapping onto a concrete architecture. We call any other execution context an *external context.* Generally, we cannot assume any compatibility between the data formats and calling conventions between the Haskell context and a given external context, except where the Haskell 98 report explicitly prescribes a specific data format.

The principal goal of a foreign function interface is to provide a programmable interface between the Haskell context and external contexts. As a result Haskell threads can access data in external contexts and invoke functions that are executed in an external context as well as vice versa. In the rest of this report, external contexts are usually identified by a calling convention.

1.4 Cross Language Type Consistency

Given that many external languages support static types, the question arises whether the consistency of Haskell types with the types of the external language can be enforced for foreign functions. Unfortunately, this is, in general, not possible without a significant investment on the part of the implementor of the Haskell system (i.e., without implementing a dedicated type checker). For example, in the case of the C calling convention, the only other approach would be to generate a C prototype from the Haskell type and leave it to the C compiler to match this prototype with the prototype that is specified in a C header file for the imported function. However, the Haskell type is lacking some information that would be required to pursue this route. In particular, the Haskell type does not contain any information as to when `const` modifiers have to be emitted.

As a consequence, this report does not require the Haskell system to check consistency with foreign types. Nevertheless, Haskell systems are encouraged to provide any cross language consistency checks that can be implemented with reasonable effort.

2 Lexical Structure

In the following, all formal grammatical definitions are based on the same notation as that defined in the Haskell 98 Report [7, Section 2.1] and we make free use of all nonterminals defined in the Haskell 98 Report. The only addition to the lexical structure of Haskell 98 [7, Section 2] is a single new reserved identifier (namely, `foreign`) and a set of special identifiers. The latter have a special meaning only within foreign declarations, but may be used as ordinary identifiers elsewhere.

The following productions are added:

reservedid → `foreign`
specialid → `export` | `safe` | `unsafe` | `ccall`
| `cplusplus` | `dotnet` | `jvm` | `stdcall`
| **system-specific calling conventions**

The special identifiers `ccall`, `cplusplus`, `dotnet`, `jvm`, and `stdcall` are defined to denote calling conventions. However, a concrete implementation of the FFI is free to support additional, system-specific calling conventions whose name is not explicitly listed here.

To refer to objects of an external C context, we introduce the following phrases:

chname → {*chchar*} `.` `h` (C header filename)
cid → *letter* {*letter* | *ascDigit*} (C identifier)
chchar → *letter* | *ascSymbol*$_{\langle \& \rangle}$
letter → *ascSmall* | *ascLarge* | `_`

The range of lexemes that are admissible for *chname* is a subset of those permitted as arguments to the `#include` directive in C. In particular, a file name *chname* must end in the suffix `.h`. The lexemes produced by *cid* coincide with those allowed as C identifiers, as specified in [4].

Identifier	Represented calling convention
`ccall`	Calling convention of the standard C compiler on a system
`cplusplus`	Calling convention of the standard C++ compiler on a system
`dotnet`	Calling convention of the .NET platform
`jvm`	Calling convention of the Java Virtual Machine
`stdcall`	Calling convention of the Win32 API (matches Pascal conventions)

Table 1: Calling conventions

3 Foreign Declarations

This section describes the extension of Haskell 98 by foreign declarations. The following production for the nonterminal *topdecl* extends the same nonterminal from the Haskell 98 Report. All other nonterminals are new.

topdecl	→	`foreign` *fdecl*	
fdecl	→	`import` *callconv* [*safety*] *impent* *var* `::` *ftype*	(define variable)
	|	`export` *callconv* *expent* *var* `::` *ftype*	(expose variable)
callconv	→	`ccall` | `stdcall` | `cplusplus` | `jvm` | `dotnet`	(calling convention)
	|	**system-specific calling conventions**	
impent	→	[*string*]	(imported external entity)
expent	→	[*string*]	(exported entity)
safety	→	`unsafe` | `safe`	

There are two flavours of foreign declarations: import and export declarations. An import declaration makes an *external entity,* i.e., a function or memory location defined in an external context, available in the Haskell context. Conversely, an export declaration defines a function of the Haskell context as an external entity in an external context. Consequently, the two types of declarations differ in that an import declaration defines a new variable, whereas an export declaration uses a variable that is already defined in the Haskell module.

The external context that contains the external entity is determined by the calling convention given in the foreign declaration. Consequently, the exact form of the specification of the external entity is dependent on both the calling convention and on whether it appears in an import declaration (as *impent*) or in an export declaration (as *expent*). To provide syntactic uniformity in the presence of different calling conventions, it is guaranteed that the description of an external entity lexically appears as a Haskell string lexeme. The only exception is where this string would be the empty string (i.e., be of the form `""`); in this case, the string may be omitted in its entirety.

3.1 Calling Conventions

The binary interface to an external entity on a given architecture is determined by a calling convention. It often depends on the programming language in which the external entity is implemented, but usually is more dependent on the system for which the external entity has been compiled.

As an example of how the calling convention is dominated by the system rather than the programming language, consider that an entity compiled to byte code for the Java Virtual Machine (JVM) [6] needs to be invoked by the rules of the JVM rather than that of the source language in which it is implemented (the entity might be implemented in Oberon, for example).

Any implementation of the Haskell 98 FFI must at least implement the C calling convention denoted by `ccall`. All other calling conventions are optional. Generally, the set of calling conventions is open, i.e., individual implementations may elect to support additional calling conventions. In addition to `ccall`, Table 1 specifies a range of identifiers for common calling conventions. Implementations need not implement all of these conventions, but if any is implemented, it must use the listed name. For any other calling convention, implementations are free to choose a suitable name.

The present report does only define the semantics of the calling conventions `ccall` and `stdcall`. Later versions of the report are expected to cover more calling conventions.

It should be noted that the code generated by a Haskell system to implement a particular calling convention may vary widely with the target code of that system. For example, the calling convention `jvm` will be trivial to implement for a Haskell compiler generating Java code, whereas for a Haskell compiler generating C code, the Java Native Interface (JNI) [5] has to be targeted.

3.2 Foreign Types

The following types constitute the set of *basic foreign types*:

- `Char`, `Int`, `Double`, `Float`, and `Bool` as exported by the Haskell 98 `Prelude` as well as
- `Int8`, `Int16`, `Int32`, `Int64`, `Word8`, `Word16`, `Word32`, `Word64`, `Ptr a`, `FunPtr a`, and `StablePtr a`, for any type `a`, as exported by `Foreign` (Section 5.1).

A Haskell system that implements the FFI needs to be able to pass these types between the Haskell and the external context as function arguments and results.

Foreign types are produced according to the following grammar:

$$\begin{array}{lcl}
\mathit{ftype} & \rightarrow & \mathit{frtype} \\
 & | & \mathit{fatype}\ \texttt{->}\ \mathit{ftype} \\
\mathit{frtype} & \rightarrow & \mathit{fatype} \\
 & | & \texttt{()} \\
\mathit{fatype} & \rightarrow & \mathit{qtycon}\ \ \mathit{atype}_1\ \ldots\ \mathit{atype}_k \quad (k \geq 0)
\end{array}$$

A foreign type is the Haskell type of an external entity. Only a subset of Haskell's types are permissible as foreign types, as only a restricted set of types can be canonically transferred between the Haskell context and an external context. A foreign type generally has the form

$$\mathit{at}_1\ \texttt{->}\ \cdots\ \texttt{->}\ \mathit{at}_n\ \texttt{->}\ \mathit{rt}$$

where $n \geq 0$. It implies that the arity of the external entity is n.

The argument types at_i produced by *fatype* must be *marshallable foreign types;* that is, each at_i is either (1) a basic foreign type or (2) a type synonym or renamed datatype of a marshallable foreign type. Moreover, the result type rt produced by *frtype* must be a *marshallable foreign result type;* that is, it is either a marshallable foreign type, the type `()`, or a type matching `Prelude.IO` t, where t is a marshallable foreign type or `()`.

External functions are generally strict in all arguments.

3.3 Import Declarations

Generally, an import declaration has the form

`foreign import` c e v `::` t

which declares the variable v of type t to be defined externally. Moreover, it specifies that v is evaluated by executing the external entity identified by the string e using calling convention c. The precise form of e depends on the calling convention and is detailed in Section 4. If a variable v is defined by an import declaration, no other top-level declaration for v is allowed in the same module. For example, the declaration

```
foreign import ccall "string.h strlen" cstrlen :: Ptr CChar -> IO CSize
```

introduces the function `cstrlen`, which invokes the external function `strlen` using the standard C calling convention. Some external entities can be imported as pure functions; for example,

```
foreign import ccall "math.h sin" sin :: CDouble -> CDouble.
```

Such a declaration asserts that the external entity is a true function; i.e., when applied to the same argument values, it always produces the same result.

Whether a particular form of external entity places a constraint on the Haskell type with which it can be imported is defined in Section 4. Although, some forms of external entities restrict the set of Haskell types that are permissible, the system can generally not guarantee the consistency between the Haskell type given in an import declaration and the argument and result types of the external entity. It is the responsibility of the programmer to ensure this consistency.

Optionally, an import declaration can specify, after the calling convention, the safety level that should be used when invoking an external entity. A `safe` call is less efficient, but guarantees to leave the Haskell system in a state that allows callbacks from the external code. In contrast, an `unsafe` call, while carrying less overhead, must not trigger a callback into the Haskell system. If it does, the system behaviour is undefined. The default for an invocation is to be `safe`. Note that a callback into the Haskell system implies that a garbage collection might be triggered after an external entity was called, but before this call returns. Consequently, objects other than stable pointers (cf. Section 5.6) may be moved or garbage collected by the storage manager.

3.4 Export Declarations

The general form of export declarations is

`foreign export` $c\ e\ v$ `::` t

Such a declaration enables external access to v, which may be a value, field name, or class method that is declared on the top-level of the same module or imported. Moreover, the Haskell system defines the external entity described by the string e, which may be used by external code using the calling convention c; an external invocation of the external entity e is translated into evaluation of v. The type t must be an instance of the type of v. For example, we may have

```
foreign export ccall "addInt"   (+) :: Int   -> Int   -> Int
foreign export ccall "addFloat" (+) :: Float -> Float -> Float
```

If an evaluation triggered by an external invocation of an exported Haskell value returns with an exception, the system behaviour is undefined. Thus, Haskell exceptions have to be caught within Haskell and explicitly marshalled to the foreign code.

4 Specification of External Entities

Each foreign declaration has to specify the external entity that is accessed or provided by that declaration. The syntax and semantics of the notation that is required to uniquely determine an external entity depends heavily on the calling convention by which this entity is accessed. For example, for the calling convention `ccall`, a global label is sufficient. However, to uniquely identify a method in the calling convention `jvm`, type information has to be provided. For the latter, there is a choice between the Java source-level syntax of types and the syntax expected by JNI—but, clearly, the syntax of the specification of an external entity depends on the calling convention and may be non-trivial.

Consequently, the FFI does not fix a general syntax for denoting external entities, but requires both *impent* and *expent* to take the form of a Haskell *string* literal. The formation rules for the values of these strings depend on the calling convention and a Haskell system implementing a particular calling convention will have to parse these strings in accordance with the calling convention.

Defining *impent* and *expent* to take the form of a *string* implies that all information that is needed to statically analyse the Haskell program is separated from the information needed to generate the code interacting with the foreign language. This is, in particular, helpful for tools processing Haskell source code. When ignoring the entity information provided by *impent* or *expent*, foreign import and export declarations are still sufficient to infer identifier definition and use information as well as type information.

For more complex calling conventions, there is a choice between the user-level syntax for identifying entities (e.g., Java or C++) and the system-level syntax (e.g., the type syntax of JNI or mangled C++, respectively). If such a choice exists, the user-level syntax is preferred. Not only because it is more user friendly, but also because the system-level syntax may not be entirely independent of the particular implementation of the foreign language.

The following defines the syntax for specifying external entities and their semantics for the calling conventions `ccall` and `stdcall`. Other calling conventions from Table 1 are expected to be defined in future versions of this report.

4.1 Standard C Calls

The following defines the structure of external entities for foreign declarations under the `ccall` calling convention for both import and export declarations separately. Afterwards additional constraints on the type of foreign functions are defined.

The FFI covers only access to C functions and global variables. There are no mechanisms to access other entities of C programs. In particular, there is no support for accessing pre-processor symbols from Haskell, which includes `#define`d constants. Access from Haskell to such entities is the domain of language-specific tools, which provide added convenience over the plain FFI as defined in this report.

4.1.1 Import Declarations

For import declarations, the syntax for the specification of external entities under the `ccall` calling convention is as follows:

impent → `"` [`static`] [*chname*] [`&`] [*cid*] `"` (static function or address)
| `" dynamic "` (stub factory importing addresses)
| `" wrapper "` (stub factory exporting thunks)

The first alternative either imports a static function *cid* or, if `&` precedes the identifier, a static address. If *cid* is omitted, it defaults to the name of the imported Haskell variable. The optional filename *chname* specifies a C header file, where the intended meaning is that the header file declares the C entity identified by *cid*. In particular, when the Haskell system compiles Haskell to C code, the directive

`#include "`*chname*`"`

needs to be placed into any generated C file that refers to the foreign entity before the first occurrence of that entity in the generated C file.

The second and third alternative, identified by the keywords `dynamic` and `wrapper`, respectively, import stub functions that have to be generated by the Haskell system. In the case of `dynamic`, the stub converts C function pointers into Haskell functions; and conversely, in the case of `wrapper`, the stub converts Haskell thunks to C function pointers. If neither of the specifiers `static`, `dynamic`, or `wrapper` is given, `static` is assumed. The specifier `static` is nevertheless needed to import C routines that are named `dynamic` or `wrapper`.

It should be noted that a static foreign declaration that does not import an address (i.e., where `&` is not used in the specification of the external entity) always refers to a C function, even if the Haskell type is non-functional. For example,

```
foreign import ccall foo :: CInt
```

refers to a pure C function `foo` with no arguments that returns an integer value. Similarly, if the type is `IO CInt`, the declaration refers to an impure nullary function. If a Haskell program needs to access a C variable `bar` of integer type,

```
foreign import ccall "&" bar :: Ptr CInt
```

must be used to obtain a pointer referring to the variable. The variable can be read and updated using the routines provided by the module `Storable` (cf. Section 5.7).

4.1.2 Export Declarations

External entities in *ccall* export declarations are of the form

expent → " [*cid*] "

The optional C identifier *cid* defines the external name by which the exported Haskell variable is accessible in C. If it is omitted, the external name defaults to the name of the exported Haskell variable.

4.1.3 Constraints on Foreign Function Types

In the case of import declaration, there are, depending on the kind of import declaration, constraints regarding the admissible Haskell type that the variable defined in the import may have. These constraints are specified in the following.

Static Functions. A static function can be of any foreign type; in particular, the result type may or may not be in the IO monad. If a function that is not pure is not imported in the IO monad, the system behaviour is undefined. Generally, no check for consistency with the C type of the imported label is performed.

As an example, consider

```
foreign import ccall "static stdlib.h" system :: Ptr CChar -> IO CInt
```

This declaration imports the `system()` function whose prototype is available from `stdlib.h`.

Static addresses. The type of an imported address is constrained to be of the form `Ptr` *a* or `FunPtr` *a*, where *a* can be any type.

As an example, consider

```
foreign import ccall "errno.h &errno" errno :: Ptr CInt
```

It imports the address of the variable `errno`, which is of the C type `int`.

Dynamic import. The type of a *dynamic* stub has to be of the form `(FunPtr` *ft*`) ->` *ft*, where *ft* may be any foreign type.

As an example, consider

```
foreign import ccall "dynamic"
  mkFun :: FunPtr (CInt -> IO ()) -> (CInt -> IO ())
```

The stub factory `mkFun` converts any pointer to a C function that gets an integer value as its only argument and does not have a return value into a corresponding Haskell function.

Dynamic wrapper. The type of a *wrapper* stub has to be of the form *ft* `-> IO (FunPtr` *ft*`)`, where *ft* may be any foreign type.

As an example, consider

```
foreign import ccall "wrapper"
  mkCallback :: IO () -> IO (FunPtr (IO ()))
```

The stub factory `mkCallback` turns any Haskell computation of type `IO ()` into a C function pointer that can be passed to C routines, which can call back into the Haskell context by invoking the referenced function.

4.1.4 Specification of Header Files

A C header specified in an import declaration is always included by `#include "`*chname*`"`. There is no explicit support for `#include <`*chname*`>` style inclusion. The ISO C99 [3] standard guarantees that any search path that would be used for a `#include <`*chname*`>` is also used for `#include "`*chname*`"` and it is guaranteed that these paths are searched after all paths that are unique to `#include "`*chname*`"`. Furthermore, we require that *chname* ends on `.h` to make parsing of the specification of external entities unambiguous.

The specification of include files has been kept to a minimum on purpose. Libraries often require a multitude of include directives, some of which may be system-dependent. Any design that attempts to cover all possible configurations would introduce significant complexity. Moreover, in the current design, a custom include file can be specified that uses the standard C preprocessor features to include all relevant headers.

Header files have no impact on the semantics of a foreign call, and whether an implementation uses the header file or not is implementation-defined. However, as some implementations may require a header file that supplies a correct prototype for external functions in order to generate correct code, portable FFI code must include suitable header files.

4.1.5 C Argument Promotion

The argument passing conventions of C are dependant on whether a function prototype for the called functions is in scope at a call site. In particular, if no function prototype is in scope, *default argument promotion* is applied to integral and floating types. In general, it cannot be expected from a Haskell system that it is aware of whether a given C function was compiled with or without a function prototype being in scope. For the sake of portability, we thus require that a Haskell system generally implements calls to C functions as well as C stubs for Haskell functions as if a function prototype for the called function is in scope.

This convention implies that the onus for ensuring the match between C and Haskell code is placed on the FFI user. In particular, when a C function that was compiled without a prototype is called from Haskell, the Haskell signature at the corresponding `foreign import` declaration must use the types *after* argument promotion. For example, consider the following C function definition, which lacks a prototype:

```
void foo (a)
float a;
{
  ...
}
```

The lack of a prototype implies that a C compiler will apply default argument promotion to the parameter `a`, and thus, `foo` will expect to receive a value of type `double`, *not* `float`. Hence, the correct `foreign import` declaration is

```
foreign import ccall foo :: Double -> IO ()
```

In contrast, a C function compiled with the prototype

```
void foo (float a);
```

requires

```
foreign import ccall foo :: Float -> IO ()
```

A similar situation arises in the case of `foreign export` declarations that use types that would be altered under the C default argument promotion rules. When calling such Haskell functions from C, a function prototype matching the signature provided in the `foreign export` declaration must be in scope; otherwise, the C compiler will erroneously apply the promotion rules to all function arguments.

Note that for a C function defined to a accept a variable number of arguments, all arguments beyond the explicitly typed arguments suffer argument promotion. However, because C permits the calling convention to be different for such functions; a Haskell system will, in general, not be able to make use of variable argument functions. Hence, their use is deprecated in portable code.

4.2 Win32 API Calls

The specification of external entities under the `stdcall` calling convention is identical to that for standard C calls. The two calling conventions only differ in the generated code.

5 Marshalling

In addition to the language extension discussed in previous sections, the FFI includes a set of standard libraries, which ease portable use of foreign functions as well as marshalling of compound structures. Generally, the marshalling of Haskell structures into a foreign representation and vice versa can be implemented in either Haskell or the foreign language. At least where the foreign language is at a significantly lower level, e.g. C, there are good reasons for doing the marshalling in Haskell:

- Haskell's lazy evaluation strategy would require any foreign code that attempts to access Haskell structures to force the evaluation of these structures before accessing them. This would lead to complicated code in the foreign language, but does not need any extra consideration when coding the marshalling in Haskell.
- Despite the fact that marshalling code in Haskell tends to look like C in Haskell syntax, the strong type system still catches many errors that would otherwise lead to difficult-to-debug runtime faults.
- Direct access to Haskell heap structures from a language like C—especially, when marshalling from C to Haskell, i.e., when Haskell structures are created—carries the risk of corrupting the heap, which usually leads to faults that are very hard to debug.

Consequently, the Haskell FFI emphasises Haskell-side marshalling.

The interface to the marshalling libraries is provided by the module `Foreign` plus a language-dependent module per supported language. In particular, the standard requires the availability of the module `CForeign`, which simplifies portable interfacing with external C code. Language-dependent modules, such as `CForeign`, generally provide Haskell types representing the basic types of the foreign language using a representation that is compatible with the foreign types as implemented by the default implementation of the foreign language on the present architecture. This is especially important for languages where the standard leaves some aspects of the implementation of basic types open. For example, in C, the size of the various integral types is not fixed. Thus, to represent C interfaces faithfully in Haskell, for each integral type in C, we need to have an integral type in Haskell that is guaranteed to have the same size as the corresponding C type.

In the following, the interface of the language independent support is defined. The interface for C-specific support is discussed in Section 6.

5.1 `Foreign`

The module `Foreign` combines the interfaces of all modules providing language-independent marshalling support. These modules are `Bits`, `Int`, `Word`, `Ptr`, `ForeignPtr`, `StablePtr`, `Storable`, `MarshalAlloc`, `MarshalArray`, `MarshalError`, and `MarshalUtils`.

Sometimes an external entity is a pure function, except that it passes arguments and/or results via pointers. To permit the packaging of such entities as pure functions, `Foreign` provides the following primitive:

`unsafePerformIO :: IO a -> a`

Return the value resulting from executing the `IO` action. This value should be independent of the environment; otherwise, the system behaviour is undefined.

If the `IO` computation wrapped in `unsafePerformIO` performs side effects, then the relative order in which those side effects take place (relative to the main `IO` trunk, or other calls to `unsafePerformIO`) is indeterminate. Moreover, the side effects may be performed several times or not at all, depending on lazy evaluation and whether the compiler unfolds an enclosing definition.

Great care should be exercised in the use of this primitive. Not only because of the danger of introducing side effects, but also because `unsafePerformIO` may compromise typing; to avoid this, the programmer should ensure that the result of `unsafePerformIO` has a monomorphic type.

5.2 `Bits`

This module provides functions implementing typical bit operations overloaded for the standard integral types `Int` and `Integer` as well as the types provided by the modules `Int` and `Word` in Section 5.3. The overloading is implemented via a new type class `Bits`, which is a subclass of `Num` and has the following member functions:

`(.&.), (.|.), xor :: Bits a => a -> a -> a`

Implement bitwise conjunction, disjunction, and exclusive or. The infix operators have the following precedences:

```
infixl 7 .&.
infixl 6 `xor`
infixl 5 .|.
```

`complement :: Bits a => a -> a`

Calculate the bitwise complement of the argument.

`shift, rotate :: Bits a => a -> Int -> a`

Shift or rotate the bit pattern to the left for a positive second argument and to the right for a negative argument. The function `shift` performs sign extension on signed number types; i.e., right shifts fill the top bits with 1 if the number is negative and with 0 otherwise. These operators have the following precedences as infix operators:

```
infixl 8 `shift`, `rotate`
```

For unbounded types (e.g., `Integer`), `rotate` is equivalent to `shift`. An instance can define either this unified `rotate` or `rotateL` and `rotateR`, depending on which is more convenient for the type in question.

`bit :: Bits a => Int -> a`

Obtain a value where only the nth bit is set.

`setBit, clearBit, complementBit :: a -> Int -> a`

Set, clear, or complement the bit at the given position.

`testBit :: Bits a => a -> Int -> Bool`

Check whether the nth bit of the first argument is set.

```
bitSize  :: Bits a => a -> Int
isSigned :: Bits a => a -> Bool
```

Respectively, query the number of bits of values of type `a` and whether these values are signed. These functions never evaluate their argument. The function `bitSize` is undefined for unbounded types (e.g., `Integer`).

```
shiftL,  shiftR  :: Bits a => a -> Int -> a
rotateL, rotateR :: Bits a => a -> Int -> a
```

The functions `shiftR` and `rotateR` are synonyms for `shift` and `rotate`; `shiftL` and `rotateL` negate the second argument. These operators have the following precedences as infix operators:

```
infixl 8 `shiftL`, `shiftR`, `rotateL`, `rotateR`
```

Bits are numbered from 0 with bit 0 being the least significant bit. A minimal complete definition of the type class `Bits` must include definitions for the following functions: `(.&.)`, `(.|.)`, `xor`, `complement`, `shift`, `rotate`, `bitSize`, and `isSigned`.

5.3 `Int` and `Word`

The two modules `Int` and `Word` provide the following signed and unsigned integral types of fixed size:

Size in bits	Signed	Unsigned
8	`Int8`	`Word8`
16	`Int16`	`Word16`
32	`Int32`	`Word32`
64	`Int64`	`Word64`

For these integral types, the modules `Int` and `Word` export class instances for the class `Bits` and all type classes for which `Int` has an instance in the Haskell 98 Prelude and standard libraries. The constraints on the implementation of these instances are also the same as those outlined for `Int` in the Haskell Report. There is, however, the additional constraint that all arithmetic on the fixed-sized types is performed modulo 2^n.

5.4 `Ptr`

The module `Ptr` provides typed pointers to foreign entities. We distinguish two kinds of pointers: pointers to data and pointers to functions. It is understood that these two kinds of pointers may be represented differently as they may be references to data and text segments, respectively.

5.4.1 Data Pointers

The interface defining data pointers and associated operations is as follows:

`data Ptr a`

A value of type `Ptr a` represents a pointer to an object, or an array of objects, which may be marshalled to or from Haskell values of type `a`. The type `a` will normally be an instance of class `Storable` (see Section 5.7), which provides the necessary marshalling operations.

Instances for the classes `Eq`, `Ord`, and `Show` are provided.

`nullPtr :: Ptr a`

The constant `nullPtr` contains a distinguished value of `Ptr` that is not associated with a valid memory location.

`castPtr :: Ptr a -> Ptr b`

The `castPtr` function casts a pointer from one type to another.

`plusPtr :: Ptr a -> Int -> Ptr b`

Advances the given address by the given offset in bytes.

`alignPtr :: Ptr a -> Int -> Ptr a`

Given an arbitrary address and an alignment constraint, `alignPtr` yields an address, the same or next higher, that fulfills the alignment constraint. An alignment constraint `x` is fulfilled by any address divisible by `x`. This operation is idempotent.

`minusPtr :: Ptr a -> Ptr b -> Int`

Compute the offset required to get from the first to the second argument. We have

```
p2 == p1 `plusPtr` (p2 `minusPtr` p1)
```

It should be noted that the use of `Int` for pointer differences essentially forces any implementation to represent `Int` in as many bits as used in the representation of pointer values.

5.4.2 Function Pointers

The interface defining function pointers and associated operations is as follows:

`data FunPtr a`

A value of type `FunPtr a` is a pointer to a piece of code. It may be the pointer to a C function or to a Haskell function created using a wrapper stub as outlined in Section 4.1. For example,

```
type Compare = Int -> Int -> Bool
foreign import ccall "wrapper"
  mkCompare :: Compare -> IO (FunPtr Compare)
```

Instances for the classes `Eq`, `Ord`, and `Show` are provided.

`nullFunPtr :: FunPtr a`

The constant `nullFunPtr` contains a distinguished value of `FunPtr` that is not associated with a valid memory location.

`castFunPtr :: FunPtr a -> FunPtr b`

Cast a `FunPtr` to a `FunPtr` of a different type.

`freeHaskellFunPtr :: FunPtr a -> IO ()`

Release the storage associated with the given `FunPtr`, which must have been obtained from a wrapper stub. This should be called whenever the return value from a foreign import wrapper function is no longer required; otherwise, the storage it uses will leak.

Moreover, there are two functions that are only valid on architectures where data and function pointers range over the same set of addresses. Only where bindings to external libraries are made whose interface already relies on this assumption, should the use of `castFunPtrToPtr` and `castPtrToFunPtr` be considered; otherwise, it is recommended to avoid using these functions.

`castFunPtrToPtr :: FunPtr a -> Ptr b`
`castPtrToFunPtr :: Ptr a -> FunPtr b`

These two functions cast `Ptr`s to `FunPtr`s and vice versa.

5.5 ForeignPtr

The type `ForeignPtr` represents references to objects that are maintained in a foreign language, i.e., objects that are not part of the data structures usually managed by the Haskell storage manager. The type `ForeignPtr` is parameterised in the same way as `Ptr` (cf. Section 5.4), but in contrast to vanilla memory references of type `Ptr`, `ForeignPtr`s may be associated with finalizers. A finalizer is a routine that is invoked when the Haskell storage manager detects that—within the Haskell heap and stack—there are no more references left that are pointing to the `ForeignPtr`. Typically, the finalizer will free the resources bound by the foreign object. Finalizers are generally implemented in the foreign language[1] and have either of the following two Haskell types:

```
type FinalizerPtr        a = FunPtr (           Ptr a -> IO ())
type FinalizerEnvPtr env a = FunPtr (Ptr env -> Ptr a -> IO ())
```

A foreign finalizer is represented as a pointer to a C function of type `Ptr a -> IO ()` or a C function of type `Ptr env -> Ptr a -> IO ()`, where `Ptr env` represents an optional environment passed to the finalizer on invocation. That is, a foreign finalizer attached to a finalized pointer `ForeignPtr a` gets the finalized pointer in the form of a raw pointer of type `Ptr a` as an argument when it is invoked. In addition, a foreign finalizer of type `FinalizerEnvPtr env a` also gets an environment pointer of type `Ptr env`. There is no guarantee on how soon the finalizer is executed after the last reference to the associated foreign pointer was dropped; this depends on the details of the Haskell storage manager. The only guarantee is that the finalizer runs before the program

---

[1] Finalizers in Haskell cannot be safely realised without requiring support for concurrency [1].

terminates. Whether a finalizer may call back into the Haskell system is system dependent. Portable code may not rely on such callbacks.

Foreign finalizers that expect an environment are a means to model closures in languages that do not support them natively, such as C. They recover part of the convenience lost by requiring finalizers to be defined in the foreign languages rather than in Haskell.

The data type `ForeignPtr` and associated operations have the following signature and purpose:

`data ForeignPtr a`

A value of type `ForeignPtr a` represents a pointer to an object, or an array of objects, which may be marshalled to or from Haskell values of type `a`. The type `a` will normally be an instance of class `Storable` (see Section 5.7), which provides the marshalling operations.

Instances for the classes `Eq`, `Ord`, and `Show` are provided. Equality and ordering of two foreign pointers are the same as for the plain pointers obtained with `unsafeForeignPtrToPtr` from those foreign pointers.

`newForeignPtr_ :: Ptr a -> IO (ForeignPtr a)`

Turn a plain memory reference into a foreign pointer that may be associated with finalizers by using `addForeignPtrFinalizer`.

`newForeignPtr :: FinalizerPtr a -> Ptr a -> IO (ForeignPtr a)`

This is a convenience function that turns a plain memory reference into a foreign pointer and immediately adds a finalizer. It is defined as

```
newForeignPtr finalizer ptr =
  do
    fp <- newForeignPtr_ ptr
    addForeignPtrFinalizer finalizer fp
    return fp
```

`newForeignPtrEnv :: FinalizerEnvPtr env a -> Ptr env -> Ptr a -> IO (ForeignPtr a)`

This variant of `newForeignPtr` adds a finalizer that expects an environment in addition to the finalized pointer. The environment that will be passed to the finalizer is fixed by the second argument to `newForeignPtrEnv`.

`addForeignPtrFinalizer :: FinalizerPtr a -> ForeignPtr a -> IO ()`

Add a finalizer to the given foreign pointer. All finalizers associated with a single foreign pointer are executed in the opposite order of their addition—i.e., the finalizer added last will be executed first.

```
addForeignPtrFinalizerEnv :: FinalizerEnvPtr env a -> Ptr env -> ForeignPtr a
                          -> IO ()
```

Add a finalizer that expects an environment to an existing foreign pointer.

`mallocForeignPtr :: Storable a => IO (ForeignPtr a)`

Allocate a block of memory that is sufficient to hold values of type `a`. The size of the memory area is determined by the function `Storable.sizeOf` (Section 5.7). This corresponds to `MarshalAlloc.malloc` (Section 5.8), but automatically attaches a finalizer that frees the block of memory as soon as all references to that block of of memory have been dropped. It is *not* guaranteed that the block of memory was allocated by `MarshalAlloc.malloc`; so, `MarshalAlloc.realloc` must not be applied to the resulting pointer.

`mallocForeignPtrBytes :: Int -> IO (ForeignPtr a)`

Allocate a block of memory of the given number of bytes with a finalizer attached that frees the block of memory as soon as all references to that block of memory have been dropped. As for `mallocForeignPtr`, `MarshalAlloc.realloc` must not be applied to the resulting pointer.

`mallocForeignPtrArray  :: Storable a => Int -> IO (ForeignPtr a)` `mallocForeignPtrArray0 :: Storable a => Int -> IO (ForeignPtr a)`

These functions correspond to `MarshalArray`'s `mallocArray` and `mallocArray0`, respectively, but yield a memory area that has a finalizer attached that releases the memory area. As with the previous two functions, it is not guaranteed that the block of memory was allocated by `MarshalAlloc.malloc`.

`withForeignPtr :: ForeignPtr a -> (Ptr a -> IO b) -> IO b`

This is a way to obtain the pointer living inside a foreign pointer. This function takes a function which is applied to that pointer. The resulting `IO` action is then executed. The foreign pointer is kept alive at least during the whole action, even if it is not used directly inside. Note that it is not safe to return the pointer from the action and use it after the action completes. All uses of the pointer should be inside the `withForeignPtr` bracket.

More precisely, the foreign pointer may be finalized after `withForeignPtr` is finished if the first argument was the last occurrence of that foreign pointer. Finalisation of the foreign pointer might render the pointer that is passed to the function useless. Consequently, this pointer cannot be used safely anymore after the `withForeignPtr` is finished, unless the function `touchForeignPtr` is used to explicitly keep the foreign pointer alive.

This function is normally used for marshalling data to or from the object pointed to by the `ForeignPtr`, using the operations from the `Storable` class.

`unsafeForeignPtrToPtr :: ForeignPtr a -> Ptr a`

Extract the pointer component of a foreign pointer. This is a potentially dangerous operation. If the argument to `unsafeForeignPtrToPtr` is the last usage occurrence of the given foreign pointer, then its finalizer(s) will be run, which potentially invalidates the plain pointer just obtained. Hence, `touchForeignPtr` must be used wherever it has to be guaranteed that the pointer lives on—i.e., has another usage occurrence.

It should be noticed that this function does not need to be monadic when used in combination with `touchForeignPtr`. Until the `unsafeForeignPtrToPtr` is executed, the thunk representing the suspended call keeps the foreign pointer alive. Afterwards, the `touchForeignPtr` keeps the pointer alive.

To avoid subtle coding errors, hand written marshalling code should preferably use the function `withForeignPtr` rather than `unsafeForeignPtrToPtr` and `touchForeignPtr`. However, the later routines are occasionally preferred in tool-generated marshalling code.

`touchForeignPtr :: ForeignPtr a -> IO ()`

Ensure that the foreign pointer in question is alive at the given place in the sequence of `IO` actions. In particular, `withForeignPtr` does a `touchForeignPtr` after it executes the user action.

This function can be used to express liveness dependencies between `ForeignPtr`s: For example, if the finalizer for one `ForeignPtr` touches a second `ForeignPtr`, then it is ensured that the second `ForeignPtr` will stay alive at least as long as the first. This can be useful when you want to manipulate interior pointers to a foreign structure: You can use `touchForeignPtr` to express the requirement that the exterior pointer must not be finalized until the interior pointer is no longer referenced.

`castForeignPtr :: ForeignPtr a -> ForeignPtr b`

Cast a `ForeignPtr` parameterised by one type into another type.

5.6 StablePtr

A *stable pointer* is a reference to a Haskell expression that is guaranteed not to be affected by garbage collection, i.e., it will neither be deallocated nor will the value of the stable pointer itself change during garbage collection (ordinary references may be relocated during garbage collection). Consequently, stable pointers can be passed to foreign code, which can treat it as an opaque reference to a Haskell value.

The data type and associated operations have the following signature and purpose:

`data StablePtr a`
Values of this type represent a stable reference to a Haskell value of type `a`.

`newStablePtr :: a -> IO (StablePtr a)`
Create a stable pointer referring to the given Haskell value.

`deRefStablePtr :: StablePtr a -> IO a`
Obtain the Haskell value referenced by a stable pointer, i.e., the same value that was passed to the corresponding call to `makeStablePtr`. If the argument to `deRefStablePtr` has already been freed using `freeStablePtr`, the behaviour of `deRefStablePtr` is undefined.

`freeStablePtr :: StablePtr a -> IO ()`
Dissolve the association between the stable pointer and the Haskell value. Afterwards, if the stable pointer is passed to `deRefStablePtr` or `freeStablePtr`, the behaviour is undefined. However, the stable pointer may still be passed to `castStablePtrToPtr`, but the `Ptr ()` value returned by `castStablePtrToPtr`, in this case, is undefined (in particular, it may be `Ptr.nullPtr`). Nevertheless, the call to `castStablePtrToPtr` is guaranteed not to diverge.

`castStablePtrToPtr :: StablePtr a -> Ptr ()`
Coerce a stable pointer to an address. No guarantees are made about the resulting value, except that the original stable pointer can be recovered by `castPtrToStablePtr`. In particular, the address may not refer to an accessible memory location and any attempt to pass it to the member functions of the class `Storable` (Section 5.7) leads to undefined behaviour.

`castPtrToStablePtr :: Ptr () -> StablePtr a`
The inverse of `castStablePtrToPtr`, i.e., we have the identity

```
sp == castPtrToStablePtr (castStablePtrToPtr sp)
```

for any stable pointer `sp` on which `freeStablePtr` has not been executed yet. Moreover, `castPtrToStablePtr` may only be applied to pointers that have been produced by `castStablePtrToPtr`.

It is important to free stable pointers that are no longer required by using `freeStablePtr`. Otherwise, the object referenced by the stable pointer will be retained in the heap.

5.7 Storable

To code marshalling in Haskell, Haskell data structures need to be translated into the binary representation of a corresponding data structure of the foreign language and vice versa. To this end, the module `Storable` provides routines that manipulate primitive data types stored in unstructured memory blocks. The class `Storable` is instantiated for all primitive types that can be stored in raw memory. Reading and writing these types to arbitrary memory locations is implemented by the member functions of the class. The member functions, furthermore, encompass support for computing the storage requirements and alignment restrictions of storable types.

Memory addresses are represented as values of type `Ptr a` (Section 5.4), where `a` is a storable type. The type argument to `Ptr` provides some type safety in marshalling code, as pointers to different types cannot be mixed without an explicit cast. Moreover, it assists in resolving overloading.

The class `Storable` is instantiated for all standard basic types of Haskell, the fixed size integral types of the modules `Int` and `Word` (Section 5.3), data and function pointers (Section 5.4), and stable pointers (Section 5.6). There is no instance of `Storable` for foreign pointers. The intention is to ensure that storing a foreign pointer requires an explicit cast to a plain `Ptr`, which makes it obvious that the finalizers of the foreign pointer may be invoked at this point if no other reference to the pointer exists anymore.

The signatures and behaviour of the member functions of the class `Storable` are as follows:

```
sizeOf    :: Storable a => a -> Int
alignment :: Storable a => a -> Int
```

The function `sizeOf` computes the storage requirements (in bytes) of the argument, and alignment computes the alignment constraint of the argument. An alignment constraint `x` is fulfilled by any address divisible by `x`. Both functions do not evaluate their argument, but compute the result on the basis of the type of the argument alone. We require that the size is divisible by the alignment. (Thus each element of a contiguous array of storable values will be properly aligned if the first one is.)

`peekElemOff :: Storable a => Ptr a -> Int -> IO a`

Read a value from a memory area regarded as an array of values of the same kind. The first argument specifies the start address of the array and the second the index into the array (the first element of the array has index 0).

`pokeElemOff :: Storable a => Ptr a -> Int -> a -> IO ()`

Write a value to a memory area regarded as an array of values of the same kind. The first and second argument are as for `peekElemOff`.

`peekByteOff :: Storable a => Ptr b -> Int -> IO a`

Read a value from a memory location given by a base address and byte offset from that base address.

`pokeByteOff :: Storable a => Ptr b -> Int -> a -> IO ()`

Write a value to a memory location given by a base address and offset from that base address.

`peek :: Storable a => Ptr a -> IO a`

Read a value from the given memory location.

`poke :: Storable a => Ptr a -> a -> IO ()`

Write the given value to the given memory location.

On some architectures, the `peek` and `poke` functions might require properly aligned addresses to function correctly. Thus, portable code should ensure that when peeking or poking values of some type `a`, the alignment constraint for `a`, as given by the function `alignment` is fulfilled.

A minimal complete definition of `Storable` needs to define `sizeOf`, `alignment`, one of `peek`, `peekElemOff`, or `peekByteOff`, and one of `poke`, `pokeElemOff`, and `pokeByteOff`.

5.8 `MarshalAlloc`

The module `MarshalAlloc` provides operations to allocate and deallocate blocks of raw memory (i.e., unstructured chunks of memory outside of the area maintained by the Haskell storage manager). These memory blocks are commonly used to pass compound data structures to foreign functions or to provide space in which compound result values are obtained from foreign functions. For example, Haskell lists are typically passed as C arrays to C functions; the storage space for such an array can be allocated by the following functions:

`malloc :: Storable a => IO (Ptr a)`

Allocate a block of memory that is sufficient to hold values of type `a`. The size of the memory area is determined by the function `Storable.sizeOf` (Section 5.7).

`mallocBytes :: Int -> IO (Ptr a)`

Allocate a block of memory of the given number of bytes. The block of memory is sufficiently aligned for any of the basic foreign types (see Section 3.2) that fits into a memory block of the allocated size.

`alloca :: Storable a => (Ptr a -> IO b) -> IO b`

Allocate a block of memory of the same size as `malloc`, but the reference is passed to a computation instead of being returned. When the computation terminates, free the memory area again. The operation is exception-safe; i.e., allocated memory is freed if an exception is thrown in the marshalling computation.

`allocaBytes :: Int -> (Ptr a -> IO b) -> IO b`

As `alloca`, but allocate a memory area of the given size. The same alignment constraint as for `mallocBytes` holds.

`realloc :: Storable b => Ptr a -> IO (Ptr b)`

Resize a memory area that was allocated with `malloc` or `mallocBytes` to the size needed to store values of type `b`. The returned pointer may refer to an entirely different memory area, but will be suitably aligned to hold values of type `b`. The contents of the referenced memory area will be the same as of the original pointer up to the minimum of the size of values of type `a` and `b`. If the argument to `realloc` is `Ptr.nullPtr`, `realloc` behaves like `malloc`.

`reallocBytes :: Ptr a -> Int -> IO (Ptr a)`

As `realloc`, but allocate a memory area of the given size. In addition, if the requested size is 0, `reallocBytes` behaves like `free`.

`free :: Ptr a -> IO ()`

Free a block of memory that was allocated with `malloc`, `mallocBytes`, `realloc`, `reallocBytes`, or any of the allocation functions from `MarshalArray` (see Section 5.9).

`finalizerFree :: FinalizerPtr a`

Foreign finalizer that performs the same operation as `free`.

If any of the allocation functions fails, a value of `Ptr.nullPtr` is produced. If `free` or `reallocBytes` is applied to a memory area that has been allocated with `alloca` or `allocaBytes`, the behaviour is undefined. Any further access to memory areas allocated with `alloca` or `allocaBytes`, after the computation that was passed to the allocation function has terminated, leads to undefined behaviour. Any further access to the memory area referenced by a pointer passed to `realloc`, `reallocBytes`, or `free` entails undefined behaviour.

5.9 MarshalArray

The module `MarshalArray` provides operations for marshalling Haskell lists into monolithic arrays and vice versa. Most functions come in two flavours: one for arrays terminated by a special termination element and one where an explicit length parameter is used to determine the extent of an array. The typical example for the former case are C's NUL terminated strings. However, please note that C strings should usually be marshalled using the functions provided by `CString` (Section 6.3) as the Unicode encoding has to be taken into account. All functions specifically operating on arrays that are terminated by a special termination element have a name ending on `0`—e.g., `mallocArray` allocates space for an array of the given size, whereas `mallocArray0` allocates space for one more element to ensure that there is room for the terminator.

The following functions are provided by the module:

```
mallocArray  :: Storable a => Int -> IO (Ptr a)
allocaArray  :: Storable a => Int -> (Ptr a -> IO b) -> IO b
reallocArray :: Storable a => Ptr a -> Int -> IO (Ptr a)
```

The functions behave like the functions `malloc`, `alloca`, and `realloc` provided by the module `MarshalAlloc` (Section 5.8), respectively, except that they allocate a memory area that can hold an array of elements of the given length, instead of storage for just a single element.

```
mallocArray0  :: Storable a => Int -> IO (Ptr a)
allocaArray0  :: Storable a => Int -> (Ptr a -> IO b) -> IO b
reallocArray0 :: Storable a => Ptr a -> Int -> IO (Ptr a)
```

These functions are like the previous three functions, but reserve storage space for one additional array element to allow for a termination indicator.

`peekArray :: Storable a => Int -> Ptr a -> IO [a]`

Marshal an array of the given length and starting at the address indicated by the pointer argument into a Haskell list using `Storable.peekElemOff` to obtain the individual elements. The order of elements in the list matches the order in the array.

`pokeArray :: Storable a => Ptr a -> [a] -> IO ()`
Marshal the elements of the given list into an array whose start address is determined by the first argument using `Storable.pokeElemOff` to write the individual elements. The order of elements in the array matches that in the list.

`peekArray0 :: (Storable a, Eq a) => a -> Ptr a -> IO [a]`
Marshal an array like `peekArray`, but instead of the length of the array a terminator element is specified by the first argument. All elements of the array, starting with the first element, up to, but excluding the first occurrence of an element that is equal (as determined by `==`) to the terminator are marshalled.

`pokeArray0 :: Storable a => a -> Ptr a -> [a] -> IO ()`
Marshal an array like `pokeArray`, but write a terminator value (determined by the first argument) after the last element of the list. Note that the terminator must not occur in the marshalled list if it should be possible to extract the list with `peekArray0`.

```
newArray  :: Storable a => [a] -> IO (Ptr a)
withArray :: Storable a => [a] -> (Ptr a -> IO b) -> IO b
```
These two functions combine `mallocArray` and `allocaArray`, respectively, with `pokeArray`; i.e., they allocate a memory area for an array whose length matches that of the list, and then, marshal the list into that memory area.

```
newArray0  :: Storable a => a -> [a] -> IO (Ptr a)
withArray0 :: Storable a => a -> [a] -> (Ptr a -> IO b) -> IO b
```
These two functions combine `mallocArray0` and `allocaArray0`, respectively, with the function `pokeArray0`; i.e., they allocate a memory area for an array whose length matches that of the list, and then, marshal the list into that memory area. The first argument determines the terminator.

```
copyArray :: Storable a => Ptr a -> Ptr a -> Int -> IO ()
moveArray :: Storable a => Ptr a -> Ptr a -> Int -> IO ()
```
These two functions copy entire arrays and behave like the routines `MarshalUtils.copyBytes` and `MarshalUtils.moveBytes`, respectively (Section 5.11). In particular, `moveArray` allows the source and destination array to overlap, whereas `copyArray` does not allow overlapping arrays. Both functions take a reference to the destination array as their first, and a reference to the source as their second argument. However, in contrast to the routines from `MarshalUtils` the third argument here specifies the number of array elements (whose type is specified by the parametrised pointer arguments) instead of the number of bytes to copy.

`lengthArray0 :: (Storable a, Eq a) => a -> Ptr a -> IO Int`
Determine the length of an array whose end is marked by the first occurrence of the given terminator (first argument). The length is measured in array elements (not bytes) and does not include the terminator.

`advancePtr :: Storable a => Ptr a -> Int -> Ptr a`
Advance a reference to an array by as many array elements (not bytes) as specified.

5.10 `MarshalError`

The module `MarshalError` provides language independent routines for converting error conditions of external functions into Haskell `IO` monad exceptions. It consists out of two parts. The first part extends the I/O error facilities of the `IO` module of the Haskell 98 Library Report with functionality to construct I/O errors. The second part provides a set of functions that ease turning exceptional result values into I/O errors.

5.10.1 I/O Errors

The following functions can be used to construct values of type `IOError`.

`data IOErrorType`
This is an abstract type that contains a value for each variant of `IOError`.

`mkIOError :: IOErrorType -> String -> Maybe Handle -> Maybe FilePath -> IOError`
Construct an `IOError` of the given type where the second argument describes the error location and the third and fourth argument contain the file handle and file path of the file involved in the error if applicable.

`alreadyExistsErrorType :: IOErrorType`
I/O error where the operation failed because one of its arguments already exists.

`doesNotExistErrorType :: IOErrorType`
I/O error where the operation failed because one of its arguments does not exist.

`alreadyInUseErrorType :: IOErrorType`
I/O error where the operation failed because one of its arguments is a single-use resource, which is already being used.

`fullErrorType :: IOErrorType`
I/O error where the operation failed because the device is full.

`eofErrorType :: IOErrorType`
I/O error where the operation failed because the end of file has been reached.

`illegalOperationType :: IOErrorType`
I/O error where the operation is not possible.

`permissionErrorType :: IOErrorType`
I/O error where the operation failed because the user does not have sufficient operating system privilege to perform that operation.

`userErrorType :: IOErrorType`
I/O error that is programmer-defined.

`annotateIOError :: IOError -> String -> Maybe Handle -> Maybe FilePath -> IOError`
Adds a location description and maybe a file path and file handle to an I/O error. If any of the file handle or file path is not given the corresponding value in the I/O error remains unaltered.

5.10.2 Result Value Checks

The following routines are useful for testing return values and raising an I/O exception in case of values indicating an error state.

`throwIf :: (a -> Bool) -> (a -> String) -> IO a -> IO a`
Execute the computation determined by the third argument. If the predicate provided in the first argument yields `True` when applied to the result of that computation, raise an `IO` exception that includes an error message obtained by applying the second argument to the result of the computation. If no exception is raised, the result of the computation is the result of the whole operation.

`throwIf_:: (a -> Bool) -> (a -> String) -> IO a -> IO ()`
Operate as `throwIf` does, but discard the result of the computation in any case.

```
throwIfNeg  :: (Ord a, Num a) => (a -> String) -> IO a -> IO a
throwIfNeg_ :: (Ord a, Num a) => (a -> String) -> IO a -> IO ()
```

These two functions are instances of `throwIf` and `throwIf_`, respectively, where the predicate is `(< 0)`.

`throwIfNull :: String -> IO (Ptr a) -> IO (Ptr a)`
This is an instance of `throwIf`, where the predicate is `(== Ptr.nullPtr)` and the error message is constant.

`void :: IO a -> IO ()`
Discard the result of a computation.

5.11 `MarshalUtils`

Finally, the module `MarshalUtils` provides a set of useful auxiliary routines.

```
new :: Storable a => a -> IO (Ptr a)
```

This function first applies `MarshalAlloc.malloc` (Section 5.8) to its argument, and then, stores the argument in the newly allocated memory area using `Storable.poke` (Section 5.7).

```
with :: Storable a => a -> (Ptr a -> IO b) -> IO b
```

This function is like `new`, but uses `MarshalAlloc.alloca` instead of `MarshalAlloc.malloc`.

```
fromBool :: Num a => Bool -> a
toBool   :: Num a => a -> Bool
```

These two functions implement conversions between Haskell Boolean values and numeric representations of Boolean values, where `False` is represented by `0` and `True` by any non-zero value.

```
maybeNew :: (a -> IO (Ptr a)) -> (Maybe a -> IO (Ptr a))
```

Lift a function that marshals a value of type `a` to a function that marshals a value of type `Maybe a`. In case, where the latter is `Nothing`, return `Ptr.nullPtr` (Section 5.4)

```
maybeWith :: (a -> (Ptr b -> IO c) -> IO c)-> (Maybe a -> (Ptr b -> IO c) -> IO c)
```

This function lifts a `MarshalAlloc.alloca` based marshalling function for `a` to `Maybe a`. It marshals values `Nothing` in the same way as `maybeNew`.

```
maybePeek :: (Ptr a -> IO b) -> (Ptr a -> IO (Maybe b))
```

Given a function that marshals a value stored in the referenced memory area to a value of type `b`, lift it to producing a value of type `Maybe b`. If the pointer is `Ptr.nullPtr`, produce `Nothing`.

```
copyBytes :: Ptr a -> Ptr a -> Int -> IO ()
moveBytes :: Ptr a -> Ptr a -> Int -> IO ()
```

These two functions are Haskell variants of the standard C library routines `memcpy()` and `memmove()`, respectively. As with their C counterparts, `moveBytes` allows the source and destination array to overlap, whereas `copyBytes` does not allow overlapping areas. Both functions take a reference to the destination area as their first, and a reference to the source as their second argument—i.e., the argument order is as in an assignment.

C symbol	Haskell symbol	Constraint on concrete C type
`HsChar`	`Char`	integral type
`HsInt`	`Int`	signed integral type, ≥ 30 bit
`HsInt8`	`Int8`	signed integral type, 8 bit; `int8_t` if available
`HsInt16`	`Int16`	signed integral type, 16 bit; `int16_t` if available
`HsInt32`	`Int32`	signed integral type, 32 bit; `int32_t` if available
`HsInt64`	`Int64`	signed integral type, 64 bit; `int64_t` if available
`HsWord8`	`Word8`	unsigned integral type, 8 bit; `uint8_t` if available
`HsWord16`	`Word16`	unsigned integral type, 16 bit; `uint16_t` if available
`HsWord32`	`Word32`	unsigned integral type, 32 bit; `uint32_t` if available
`HsWord64`	`Word64`	unsigned integral type, 64 bit; `uint64_t` if available
`HsFloat`	`Float`	floating point type
`HsDouble`	`Double`	floating point type
`HsBool`	`Bool`	`int`
`HsPtr`	`Ptr a`	`(void *)`
`HsFunPtr`	`FunPtr a`	`(void (*)(void))`
`HsStablePtr`	`StablePtr a`	`(void *)`

Table 2: C Interface to Basic Haskell Types

6 C-Specific Marshalling

6.1 `CForeign`

The module `CForeign` combines the interfaces of all modules providing C-specific marshalling support. The modules are `CTypes`, `CString`, and `CError`.

Every Haskell system that implements the FFI needs to provide a C header file named `HsFFI.h` that defines the C symbols listed in Tables 2 and 3. Table 2 table lists symbols that represent types together with the Haskell type that they represent and any constraints that are placed on the concrete C types that implement these symbols. When a C type `HsT` represents a Haskell type `T`, the occurrence of `T` in a foreign function declaration should be matched by `HsT` in the corresponding C function prototype. Indeed, where the Haskell system translates Haskell to C code that invokes `foreign import`ed C routines, such prototypes need to be provided and included via the header that can be specified in external entity strings for foreign C functions (cf. Section 4.1); otherwise, the system behaviour is undefined. It is guaranteed that the Haskell value `nullPtr` is mapped to `(HsPtr) NULL` in C and `nullFunPtr` is mapped to `(HsFunPtr) NULL` and vice versa.

Table 3 contains symbols characterising the range and precision of the types from Table 2. Where available, the table states the corresponding Haskell values. All C symbols, with the exception of `HS_FLOAT_ROUND` are constants that are suitable for use in `#if` preprocessing directives. Note that there is only one rounding style (`HS_FLOAT_ROUND`) and one radix (`HS_FLOAT_RADIX`), as this is all that is supported by ISO C [3].

Moreover, an implementation that does not support 64 bit integral types on the C side should implement `HsInt64` and `HsWord64` as a structure. In this case, the bounds `HS_INT64_MIN`, `HS_INT64_MAX`, and `HS_WORD64_MAX` are undefined.

In addition, to the symbols from Table 2 and 3, the header `HsFFI.h` must also contain the following prototypes:

```
void hs_init     (int *argc, char **argv[]);
void hs_exit     (void);
void hs_set_argv (int argc, char *argv[]);

void hs_perform_gc (void);
```

CPP symbol	Haskell value	Description
`HS_CHAR_MIN`	`minBound :: Char`	
`HS_CHAR_MAX`	`maxBound :: Char`	
`HS_INT_MIN`	`minBound :: Int`	
`HS_INT_MAX`	`maxBound :: Int`	
`HS_INT8_MIN`	`minBound :: Int8`	
`HS_INT8_MAX`	`maxBound :: Int8`	
`HS_INT16_MIN`	`minBound :: Int16`	
`HS_INT16_MAX`	`maxBound :: Int16`	
`HS_INT32_MIN`	`minBound :: Int32`	
`HS_INT32_MAX`	`maxBound :: Int32`	
`HS_INT64_MIN`	`minBound :: Int64`	
`HS_INT64_MAX`	`maxBound :: Int64`	
`HS_WORD8_MAX`	`maxBound :: Word8`	
`HS_WORD16_MAX`	`maxBound :: Word16`	
`HS_WORD32_MAX`	`maxBound :: Word32`	
`HS_WORD64_MAX`	`maxBound :: Word64`	
`HS_FLOAT_RADIX`	`floatRadix :: Float`	
`HS_FLOAT_ROUND`	n/a	rounding style as per [3]
`HS_FLOAT_EPSILON`	n/a	difference between 1 and the least value greater than 1 as per [3]
`HS_DOUBLE_EPSILON`	n/a	(as above)
`HS_FLOAT_DIG`	n/a	number of decimal digits as per [3]
`HS_DOUBLE_DIG`	n/a	(as above)
`HS_FLOAT_MANT_DIG`	`floatDigits :: Float`	
`HS_DOUBLE_MANT_DIG`	`floatDigits :: Double`	
`HS_FLOAT_MIN`	n/a	minimum floating point number as per [3]
`HS_DOUBLE_MIN`	n/a	(as above)
`HS_FLOAT_MIN_EXP`	`fst . floatRange :: Float`	
`HS_DOUBLE_MIN_EXP`	`fst . floatRange :: Double`	
`HS_FLOAT_MIN_10_EXP`	n/a	minimum decimal exponent as per [3]
`HS_DOUBLE_MIN_10_EXP`	n/a	(as above)
`HS_FLOAT_MAX`	n/a	maximum floating point number as per [3]
`HS_DOUBLE_MAX`	n/a	(as above)
`HS_FLOAT_MAX_EXP`	`snd . floatRange :: Float`	
`HS_DOUBLE_MAX_EXP`	`snd . floatRange :: Double`	
`HS_FLOAT_MAX_10_EXP`	n/a	maximum decimal exponent as per [3]
`HS_DOUBLE_MAX_10_EXP`	n/a	(as above)
`HS_BOOL_FALSE`	False	
`HS_BOOL_TRUE`	True	

Table 3: C Interface to Range and Precision of Basic Types

```
void hs_free_stable_ptr (HsStablePtr sp);
void hs_free_fun_ptr    (HsFunPtr fp);
```

These routines are useful for mixed language programs, where the main application is implemented in a foreign language that accesses routines implemented in Haskell. The function `hs_init()` initialises the Haskell system and provides it with the available command line arguments. Upon return, the arguments solely intended for the Haskell runtime system are removed (i.e., the values that `argc` and `argv` point to may have changed). This function must be called during program startup before any Haskell function is invoked; otherwise, the system behaviour is undefined. Conversely, the Haskell system is deinitialised by a call to `hs_exit()`. Multiple invocations of `hs_init()` are permitted, provided that they are followed by an equal number of calls to `hs_exit()` and that the first call to `hs_exit()` is after the last call to `hs_init()`. In addition to nested calls to `hs_init()`, the Haskell system may be de-initialised with `hs_exit()` and be re-initialised with `hs_init()` at a later point in time. This ensures that repeated initialisation due to multiple libraries being implemented in Haskell is covered.

The Haskell system will ignore the command line arguments passed to the second and any following calls to `hs_init()`. Moreover, `hs_init()` may be called with `NULL` for both `argc` and `argv`, signalling the absence of command line arguments.

The function `hs_set_argv()` sets the values returned by the functions `getProgName` and `getArgs` of the module `System` defined in the Haskell 98 Library Report. This function may only be invoked after `hs_init()`. Moreover, if `hs_set_argv()` is called at all, this call must precede the first invocation of `getProgName` and `getArgs`. Note that the separation of `hs_init()` and `hs_set_argv()` is essential in cases where in addition to the Haskell system other libraries that process command line arguments during initialisation are used.

The function `hs_perform_gc()` advises the Haskell storage manager to perform a garbage collection, where the storage manager makes an effort to releases all unreachable objects. This function must not be invoked from C functions that are imported `unsafe` into Haskell code nor may it be used from a finalizer.

Finally, `hs_free_stable_ptr()` and `hs_free_fun_ptr()` are the C counterparts of the Haskell functions `freeStablePtr` and `freeHaskellFunPtr`.

6.2 CTypes

The modules `CTypes` provide Haskell types that represent basic C types. They are needed to accurately represent C function prototypes, and so, to access C library interfaces in Haskell. The Haskell system is not required to represent those types exactly as C does, but the following guarantees are provided concerning a Haskell type `CT` representing a C type `t`:

- If a C function prototype has `t` as an argument or result type, the use of `CT` in the corresponding position in a foreign declaration permits the Haskell program to access the full range of values encoded by the C type; and conversely, any Haskell value for `CT` has a valid representation in C.

- `Storable.sizeOf (undefined :: CT)` will yield the same value as `sizeof (t)` in C.

- `Storable.alignment (undefined :: CT)` matches the alignment constraint enforced by the C implementation for `t`.

- `Storable.peek` and `Storable.poke` map all values of `CT` to the corresponding value of `t` and vice versa.

- When an instance of `Bounded` is defined for `CT`, the values of `minBound` and `maxBound` coincide with `t_MIN` and `t_MAX` in C.

- When an instance of `Eq` or `Ord` is defined for `CT`, the predicates defined by the type class implement the same relation as the corresponding predicate in C on `t`.

- When an instance of `Num`, `Read`, `Integral`, `Fractional`, `Floating`, `RealFrac`, or `RealFloat` is defined for `CT`, the arithmetic operations defined by the type class implement the same function as the corresponding arithmetic operations (if available) in C on `t`.
- When an instance of `Bits` is defined for `CT`, the bitwise operation defined by the type class implement the same function as the corresponding bitwise operation in C on `t`.

All types exported by `CTypes` must be represented as `newtype`s of basic foreign types as defined in Section 3.2 and the export must be abstract.

The module `CTypes` provides the following integral types, including instances for `Eq`, `Ord`, `Num`, `Read`, `Show`, `Enum`, `Storable`, `Bounded`, `Real`, `Integral`, and `Bits`:

Haskell type	Represented C type
`CChar`	`char`
`CSChar`	`signed char`
`CUChar`	`unsigned char`
`CShort`	`short`
`CUShort`	`unsigned short`
`CInt`	`int`
`CUInt`	`unsigned int`
`CLong`	`long`
`CULong`	`unsigned long`
`CLLong`	`long long`
`CULLong`	`unsigned long long`

Moreover, it provides the following floating point types, including instances for `Eq`, `Ord`, `Num`, `Read`, `Show`, `Enum`, `Storable`, `Real`, `Fractional`, `Floating`, `RealFrac`, and `RealFloat`:

Haskell type	Represented C type
`CFloat`	`float`
`CDouble`	`double`
`CLDouble`	`long double`

The module provides the following integral types, including instances for `Eq`, `Ord`, `Num`, `Read`, `Show`, `Enum`, `Storable`, `Bounded`, `Real`, `Integral`, and `Bits`:

Haskell type	Represented C type
`CPtrdiff`	`ptrdiff_t`
`CSize`	`size_t`
`CWchar`	`wchar_t`
`CSigAtomic`	`sig_atomic_t`

Moreover, it provides the following numeric types, including instances for `Eq`, `Ord`, `Num`, `Read`, `Show`, `Enum`, and `Storable`:

Haskell type	Represented C type
`CClock`	`clock_t`
`CTime`	`time_t`

And finally, the following types, including instances for `Eq` and `Storable`, are provided:

Haskell type	Represented C type
`CFile`	`FILE`
`CFpos`	`fpos_t`
`CJmpBuf`	`jmp_buf`

6.3 CString

The module `CString` provides routines marshalling Haskell into C strings and vice versa. The marshalling converts each Haskell character, representing a Unicode code point, to one or more bytes in a manner that, by default, is determined by the current locale. As a consequence, no guarantees can be made about the relative length of a Haskell string and its corresponding C string, and therefore, all routines provided by `CString` combine memory allocation and marshalling. The translation between Unicode and the encoding of the current locale may be lossy. The function `charIsRepresentable` identifies the characters that can be accurately translated; unrepresentable characters are converted to '?'.

```
type CString = Ptr CChar
```

A C string is a reference to an array of C characters terminated by NUL.

```
type CStringLen = (Ptr CChar, Int)
```

In addition to NUL-terminated strings, the module `CString` also supports strings with explicit length information in bytes.

```
peekCString    :: CString    -> IO String
peekCStringLen :: CStringLen -> IO String
```

Marshal a C string to Haskell. There are two variants of the routine, one for each supported string representation.

```
newCString    :: String -> IO CString
newCStringLen :: String -> IO CStringLen
```

Allocate a memory area for a Haskell string and marshal the string into its C representation. There are two variants of the routine, one for each supported string representation. The memory area allocated by these routines may be deallocated using `MarshalAlloc.free`.

```
withCString    :: String -> (CString    -> IO a) -> IO a
withCStringLen :: String -> (CStringLen -> IO a) -> IO a
```

These two routines operate as `newCString` and `newCStringLen`, respectively, but handle memory allocation and deallocation like `MarshalAlloc.alloca` (Section 5.8).

```
charIsRepresentable :: Char -> IO Bool
```

Determine whether the argument can be represented in the current locale.

Some C libraries require to ignore the Unicode capabilities of Haskell and treat values of type `Char` as single byte characters. Hence, the module `CString` provides a variant of the above marshalling routines that truncates character sets correspondingly. These functions should be used with care, as a loss of information can occur.

```
castCharToCChar :: Char -> CChar
castCCharToChar :: CChar -> Char
```

These two functions cast Haskell characters to C characters and vice versa while ignoring the Unicode encoding of the Haskell character. More precisely, only the first 256 character points are preserved.

```
peekCAString    :: CString    -> IO String
peekCAStringLen :: CStringLen -> IO String
newCAString     :: String -> IO CString
newCAStringLen  :: String -> IO CStringLen
withCAString    :: String -> (CString    -> IO a) -> IO a
withCAStringLen :: String -> (CStringLen -> IO a) -> IO a
```

These functions for whole-string marshalling cast Haskell characters to C characters and vice versa while ignoring the Unicode encoding of Haskell characters.

To simplify bindings to C libraries that use `wchar_t` for character sets that cannot be encoded in byte strings, the module `CString` also exports a variant of the above string marshalling routines for wide characters—i.e., for the C `wchar_t` type.[2]

---

[2] Note that if the platform defines `__STDC_ISO_10646__` then `wchar_t` characters are Unicode code points, and thus, the conversion between Haskell `Char` and `CWchar` is a simple cast. On other platforms, the translation is locale-dependent, just as for `CChar`.

```
type CWString    = Ptr CWchar
type CWStringLen = (Ptr CWchar, Int)
```

Wide character strings in a NUL-terminated version and a variant with explicit length information in number of wide characters.

```
peekCWString    :: CWString    -> IO String
peekCWStringLen :: CWStringLen -> IO String
newCWString     :: String -> IO CWString
newCWStringLen  :: String -> IO CWStringLen
withCWString    :: String -> (CWString    -> IO a) -> IO a
withCWStringLen :: String -> (CWStringLen -> IO a) -> IO a
```

String marshalling for wide character strings. The interface is the same as for byte strings.

6.4 `CError`

The module CError facilitates C-specific error handling of `errno`. In Haskell, we represent values of `errno` by

```
newtype Errno = Errno CInt
```

which has an instance for the type class `Eq`. The implementation of `Errno` is disclosed on purpose. Different operating systems and/or C libraries often support different values of `errno`. This module defines the common values, but due to the open definition of `Errno` users may add definitions which are not predefined. The predefined values are the following:

```
eOK, e2BIG, eACCES, eADDRINUSE, eADDRNOTAVAIL, eADV, eAFNOSUPPORT, eAGAIN,
  eALREADY, eBADF, eBADMSG, eBADRPC, eBUSY, eCHILD, eCOMM, eCONNABORTED,
  eCONNREFUSED, eCONNRESET, eDEADLK, eDESTADDRREQ, eDIRTY, eDOM, eDQUOT,
  eEXIST, eFAULT, eFBIG, eFTYPE, eHOSTDOWN, eHOSTUNREACH, eIDRM, eILSEQ,
  eINPROGRESS, eINTR, eINVAL, eIO, eISCONN, eISDIR, eLOOP, eMFILE, eMLINK,
  eMSGSIZE, eMULTIHOP, eNAMETOOLONG, eNETDOWN, eNETRESET, eNETUNREACH,
  eNFILE, eNOBUFS, eNODATA, eNODEV, eNOENT, eNOEXEC, eNOLCK, eNOLINK,
  eNOMEM, eNOMSG, eNONET, eNOPROTOOPT, eNOSPC, eNOSR, eNOSTR, eNOSYS,
  eNOTBLK, eNOTCONN, eNOTDIR, eNOTEMPTY, eNOTSOCK, eNOTTY, eNXIO,
  eOPNOTSUPP, ePERM, ePFNOSUPPORT, ePIPE, ePROCLIM, ePROCUNAVAIL,
  ePROGMISMATCH, ePROGUNAVAIL, ePROTO, ePROTONOSUPPORT, ePROTOTYPE,
  eRANGE, eREMCHG, eREMOTE, eROFS, eRPCMISMATCH, eRREMOTE, eSHUTDOWN,
  eSOCKTNOSUPPORT, eSPIPE, eSRCH, eSRMNT, eSTALE, eTIME, eTIMEDOUT,
  eTOOMANYREFS, eTXTBSY, eUSERS, eWOULDBLOCK, eXDEV
  :: Errno
```

The meaning of these values corresponds to that of the C constants of the same name with the leading "e" converted to upper-case.

The module `CError` provides the following functions:

```
isValidErrno :: Errno -> Bool
```

Yield `True` if the given `Errno` value is valid on the system. This implies that the `Eq` instance of `Errno` is also system dependent as it is only defined for valid values of `Errno`.

```
getErrno :: IO Errno
```

Get the current value of `errno`.

```
resetErrno :: IO ()
```

Reset `errno` to `eOK`.

```
errnoToIOError :: String -> Errno -> Maybe Handle -> Maybe String -> IOError
```

Compute a Haskell 98 I/O error based on the given `Errno` value. The first argument to the function should specify the location where the error occurred and the third and fourth can be used to specify a file handle and filename in the course of whose manipulation the error occurred. This is optional information, which can be used to improve the accuracy of error messages.

```
throwErrno :: String -> IO a
```

Apply `errnoToIOError` to the value currently returned by `getErrno`. Its first argument specifies the location—no extra information about a file handle or filename can be provided in this case.

```
throwErrnoIf  ::  (a -> Bool) -> String -> IO a -> IO a
throwErrnoIf_ ::  (a -> Bool) -> String -> IO a -> IO ()
```

Behave like `throwErrno` in case that the result of the `IO` action fulfils the predicate passed as a first argument. The second variant discards the result after error handling.

```
throwErrnoIfRetry  ::  (a -> Bool) -> String -> IO a -> IO a
throwErrnoIfRetry_ ::  (a -> Bool) -> String -> IO a -> IO ()
```

Like `throwErrnoIf` and `throwErrnoIf_`, but retry the `IO` action when it yields the error code `eINTR`—this amounts to the standard retry loop for interrupted POSIX system calls.

```
throwErrnoIfMinus1  ::  Num a => String -> IO a -> IO a
throwErrnoIfMinus1_ ::  Num a => String -> IO a -> IO ()
```

Instantiate `throwErrnoIf` and `throwErrnoIf_` with the predicate `(== -1)`.

```
throwErrnoIfMinus1Retry  ::  Num a => String -> IO a -> IO a
throwErrnoIfMinus1Retry_ ::  Num a => String -> IO a -> IO ()
```

Instantiate `throwErrnoIfRetry` and `throwErrnoIfRetry_` with the predicate `(== -1)`.

```
throwErrnoIfNull      ::  String -> IO (Ptr a) -> IO (Ptr a)
throwErrnoIfNullRetry ::  String -> IO (Ptr a) -> IO (Ptr a)
```

Instantiate `throwErrnoIf` and `throwErrnoIfRetry` with the predicate `(== Ptr.nullPtr)`.

References

[1] Hans-J. Boehm. Destructors, finalizers, and synchronization. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 262–272. ACM Press, 2003.

[2] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. The Java Series. Addison-Wesley, 1997.

[3] International Standard ISO/IEC. Programming languages – C. 9899:1999 (E).

[4] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, second edition, 1988.

[5] Sheng Liang. *The Java Native Interface: Programmer's Guide and Specification*. Addison Wesley, 1999.

[6] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.

[7] Simon Peyton Jones et al. Haskell 98 language and libraries: the revised report. *Journal of Functional Programming*, 13(1), 2003.