Moby Library Manual
(Preliminary)

The Moby Project
`moby.cs.uchicago.edu`

Last revised: October 25, 2004

Contents

Chapter 1

Introduction

This manual describes a collection of libraries for the MOBY programming language [**?**]. At present, this is a placeholder designed to support the writing of example programs. We plan to replace this library with a more comprehensive design once the MOBY system supports separate compilation.

Chapter 2

The MOBY Basis Library

The MOBY Basis library defines the standard set of types and operations that are available to all MOBY programs.

2.1 Pervasive definitions

```
type Bool
type Char
type Rune
type Int
type Long
type Integer
type Float
type Double
type Extended

type String
type ByteArray
type FloatArray
type DoubleArray
type Vector (t)
type Array (t)
type Ref (t)
type Ptr (t)

enumtype Order {Less, Equal, Greater}
datatype Option (t) { None, Some of (t) }
datatype Either (s, t) { In1 of (s), In2 of (t) }
datatype List (t) { Nil, Cons of (t, List(t)) }
```

```
tagtype Exn
tagtype System of String extends Exn
tagtype MatchFail extends Exn
tagtype BindFail extends Exn
tagtype Domain extends Exn
tagtype DivByZero of String extends Exn
tagtype Subscript of Int extends Exn
tagtype Size of Int extends Exn
tagtype Fail of String extends Exn

val ident : [t] t -> t
val curry : [s, t, u] ((s, t) -> u) -> s -> t -> u
val uncurry : [s, t, u] (s -> t -> u) -> (s, t) -> u
val timeOfDay : () -> (Int, Int)
val itod : Int -> Double
val itof : Int -> Float
val dtoi : Double -> Int
val ftoi : Float -> Int
```

2.2 Overloaded identifiers and operators

2.3 The **Bool** module

The `Bool` module provides a small collection of functions on the primitive `Bool` type. It has the following signature:

```
val eq  : (Bool, Bool) -> Bool
val neq : (Bool, Bool) -> Bool
val not : Bool -> Bool
```

The operations have the following semantics:

`eq`
: This function tests its arguments for equality.

> It serves as an instance of the overloaded **==** operator and will be moved to an internal module in the future.

`neq`
: This function tests its arguments for inequality.

> It serves as an instance of the overloaded **!=** operator and will be moved to an internal module in the future.

`not`

This function returns the logical negation of its argument. The operator **!** is bound to this function.

2.4 The **`Int`** module

```
val andb : (Int, Int) -> Int
val orb  : (Int, Int) -> Int
val xor  : (Int, Int) -> Int
val rsha : (Int, Int) -> Int
val rshl : (Int, Int) -> Int
val lsh  : (Int, Int) -> Int
val not  : Int -> Int
val add  : (Int, Int) -> Int
val sub  : (Int, Int) -> Int
val mul  : (Int, Int) -> Int
val div  : (Int, Int) -> Int
val rem  : (Int, Int) -> Int
val neg  : Int -> Int
val eq   : (Int, Int) -> Bool
val neq  : (Int, Int) -> Bool
val lt   : (Int, Int) -> Bool
val lte  : (Int, Int) -> Bool
val gt   : (Int, Int) -> Bool
val gte  : (Int, Int) -> Bool
val abs  : Int -> Int
val toString : Int -> String
val toStringUnsigned : Int -> String
val fromString : String -> Option(Int)
```

2.5 The **`Long`** module

The `Long` type is not supported in this release.

2.6 The `Float` module

```
val eq  : (Float, Float) -> Bool
val neq : (Float, Float) -> Bool
val lt  : (Float, Float) -> Bool
val lte : (Float, Float) -> Bool
val gt  : (Float, Float) -> Bool
val gte : (Float, Float) -> Bool
val add : (Float, Float) -> Float
val sub : (Float, Float) -> Float
val mul : (Float, Float) -> Float
val div : (Float, Float) -> Float
val pow : (Float, Float) -> Float
val neg : Float -> Float
val abs : Float -> Float
val toString : Float -> String
val fromString : String -> Option(Float)
```

2.7 The `Double` module

```
val eq  : (Double, Double) -> Bool
val neq : (Double, Double) -> Bool
val lt  : (Double, Double) -> Bool
val lte : (Double, Double) -> Bool
val gt  : (Double, Double) -> Bool
val gte : (Double, Double) -> Bool
val add : (Double, Double) -> Double
val sub : (Double, Double) -> Double
val mul : (Double, Double) -> Double
val div : (Double, Double) -> Double
val pow : (Double, Double) -> Double
val neg : Double -> Double
val abs : Double -> Double
val toString : Double -> String
val fromString : String -> Option(Double)
```

2.8 The `Extended` module

2.9 The `Char` module

```
val ord : Char -> Int
val chr : Int -> Char
val eq  : (Char, Char) -> Bool
val neq : (Char, Char) -> Bool
val lt  : (Char, Char) -> Bool
val lte : (Char, Char) -> Bool
val gt  : (Char, Char) -> Bool
val gte : (Char, Char) -> Bool
val isDigit : Char -> Bool
val isLower : Char -> Bool
val isUpper : Char -> Bool
val isSpace : Char -> Bool
```

2.10 The `String` module

The `String` module contains functions for computing with values of the pervasive `String` type. It has the following signature:

```
val eq : (String, String) -> Bool
val neq : (String, String) -> Bool
val length : String -> Int
val sub : (String, Int) -> Char
val concat2 : (String, String) -> String
val substring : (String, Int, Int) -> String
val implode : List(Char) -> String
```

The operations have the following semantics:

`eq`

This function tests its arguments for equality.

> It serves as an instance of the overloaded **==** operator and will be moved to an internal module in the future.

`neq`

This function tests its arguments for inequality.

> It serves as an instance of the overloaded **!=** operator and will be moved to an internal module in the future.

`length`

This function returns the length (number of characters) of its argument. The overloaded function `length` is also bound to this function.

`sub (s, i)`
: This function returns the `i`th character of the string `s`. If `i` is less than `0` or greater or equal to the length of `s`, then the `Subscript` exception is raised with `i` as its argument.

> This function is an instance of the overloaded subscript notation and will be moved to an internal module in the future.

`concat2`
: This function returns a string that is the concatenation of its two arguments.

> This function is an instance of the overloaded **+** operator and will be moved to an internal module in the future.

`substring (s, i, n)`
: This function returns a string of length `n` that is the substring of `s` starting at position `i`. If `i` is less than `0` or `i+n` is greater than the length of `s`, then the `Substring` exception is raised with `i` as its argument.

2.11 The `Vector` module

```
val length : [t] Vector(t) -> Int
val sub : [t] (Vector(t), Int) -> t
```

2.12 The `Array` module

```
val array : [t] (Int, t) -> Array(t)
val length : [t] Array(t) -> Int
val fromList : [t] List(t) -> Array(t)
```

2.13 The `ByteArray` module

```
val array : (Int, Int) -> ByteArray
val length : ByteArray -> Int
```

2.14 The `FloatArray` module

```
val array : (Int, Float) -> FloatArray
val length : FloatArray -> Int
```

2.15 The `DoubleArray` module

```
val array : (Int, Double) -> DoubleArray
val length : DoubleArray -> Int
```

2.16 The `Ref` module

```
val ref : [t] t -> Ref(t)
```

2.17 The `Option` module

```
val map : [s, t] (s -> t) -> Option(s) -> Option(t)
val apply : [t] (t -> ()) -> Option(t) -> ()
val default : [t] (Option(t), t) -> t
```

2.18 The `List` module

```
val length : [t] List(t) -> Int
val apply : [t] (t -> ()) -> List(t) -> ()
val append : [t] (List(t), List(t)) -> List(t)
```

Chapter 3

Math

The math library provides a collection of top-level math functions for both `Float` and `Double` values.

```
val floorf : Float -> Float
val cosf : Float -> Float
val sinf : Float -> Float
val tanf : Float -> Float
val acosf : Float -> Float
val asinf : Float -> Float
val atanf : Float -> Float
val atan2f : (Float, Float) -> Float
val coshf : Float -> Float
val sinhf : Float -> Float
val tanhf : Float -> Float
val acoshf : Float -> Float
val asinhf : Float -> Float
val atanhf : Float -> Float
val expf : Float -> Float
val logf : Float -> Float
val log10f : Float -> Float
val expm1f : Float -> Float
val log1pf : Float -> Float
val log2f : Float -> Float
val powf : (Float, Float) -> Float
val sqrtf : Float -> Float
val hypotf : (Float, Float) -> Float
```

```
val floor : Double -> Double
val cos : Double -> Double
val sin : Double -> Double
val tan : Double -> Double
val acos : Double -> Double
val asin : Double -> Double
val atan : Double -> Double
val atan2 : (Double, Double) -> Double
val cosh : Double -> Double
val sinh : Double -> Double
val tanh : Double -> Double
val acosh : Double -> Double
val asinh : Double -> Double
val atanh : Double -> Double
val exp : Double -> Double
val log : Double -> Double
val log10 : Double -> Double
val expm1 : Double -> Double
val log1p : Double -> Double
val log2 : Double -> Double
val pow : (Double, Double) -> Double
val sqrt : Double -> Double
val hypot : (Double, Double) -> Double
```

Chapter 4

Console I/O

The `ConsoleIO` module provides access to the "*console*" used to run command-line programs. This module is defined in the "`console-io`" library. It has the following signature:

```
val getCommandName : () -> String
val getCommandArgs : () -> List(String)
val getEnv : String -> Option(String)
val print : String -> ()
val printChr : Char -> ()
val inputLine : () -> String
```

The console I/O operations have the following semantics:

`getCommandName`
: This function returns the name of the program being executed (*i.e.*, `argv[0]` in UNIX systems).

`getCommandArgs`
: This function returns a list of the command-line arguments that were supplied by the user. This list does not include arguments used to control properties of the MOBY runtime system.

`getEnv`
: This function looks its argument in the executing program's *execution environment*. It returns `None` if the variable is not defined and returns `Some(`*s*`)` if the argument is bound to the value *s*.

`print`
: This function prints its argument on the standard output stream associated with the console.

`printChr`
: This function prints its argument on the standard output stream associated with the console.

`inputLine`
: This function reads a line of input from the standard input associated with the console. The empty string is returned on end-of-file.

Chapter 5

File I/O

The `FileIO` module provides simple access to reading and writing files. This module is defined in the "`file-io`" library. It has the following signature:

```
type InFile
type OutFile

val openIn : (String) -> Option(InFile)
val openOut : (String) -> Option(OutFile)
val appendOut : (String) -> Option(OutFile)
val closeIn : (InFile) -> ()
val closeOut : (OutFile) -> ()
val print : (OutFile, String) -> ()
val putc : (OutFile, Char) -> ()
val inputLine : (InFile) -> String
val getc : InFile -> Option(Char)
```

> Note that because of a bug in the current implementation of the **gen-mbi** tool, the `InFile` and `OutFile` types are pervasive (*i.e.*, they are not qualified by `FileIO`).

The file I/O operations have the following semantics:

`InFile`
: This is the type of an input file.

`OutFile`
: This is the type of an output file.

`openIn`
: This function opens the file specified by its argument for reading. If the file cannot be opened, then `None` is returned.

`openOut`
: This function opens the file specified by its argument for writing. If the file already exists, it is truncated. If the file cannot be opened, then `None` is returned.

`appendOut`
: This function opens the file specified by its argument for writing. If the file already exists, optput will be appended to it. If the file cannot be opened, then `None` is returned.

`closeIn`
: This function closes the input file given as its argument.

`closeOut`
: This function closes the output file given as its argument.

`print`
: This function prints its argument to the given output file.

`putc`
: This function prints its argument to the given output file.

`inputLine`
: This function reads a line of input from the given input file. The empty string is returned on end-of-file.

`getc`
: This function reads a single character of input from the given input file. The value `None` is returned on end-of-file.

Chapter 6

C interface

The C interface library provides a collection of basic types and operations for direct manipulation of C values. It is a support library for the **Charon** tool [FPR01].

6.1 Pervasive definitions

```
type LValue (t)
type VoidPtr
type CPtr(t) <: VoidPtr
type CFun(t)
type Arr(t)
type SizeOf(t)

type Signed
type Unsigned
type CChar(t)
type CShort(t)
type CInt(t)
type CLong(t)
type CLongLong(t)
type CFloat
type CDouble
type CLongDouble

type UChar = CChar(Unsigned)
type SChar = CChar(Signed)
type UShort = CShort(Unsigned)
type SShort = CShort(Signed)
type UInt = CInt(Unsigned)
type SInt = CInt(Signed)
type ULong = CLong(Unsigned)
type SLong = CLong(Signed)
type ULongLong = CLongLong(Unsigned)
type SLongLong = CLongLong(Signed)
```

6.2 Overloaded identifiers and operators

6.3 The `CTypes` module

```
val sizeOf : [t] SizeOf(t) -> Int
val null : [t] () -> CPtr(t)
val malloc : [t] (SizeOf(t)) -> CPtr(t)
val nmalloc : [t] (Int, SizeOf(t)) -> Arr(t)
val free : [t] CPtr(t) -> ()

val getSChar : LValue(SChar) -> Int
val setSChar : (LValue(SChar), Int) -> ()
val sizeOfSChar : () -> SizeOf(SChar)
val getUChar : LValue(UChar) -> Int
val setUChar : (LValue(UChar), Int) -> ()
val sizeOfUChar : () -> SizeOf(UChar)
val getSShort : LValue(SShort) -> Int
val setSShort : (LValue(SShort), Int) -> ()
val sizeOfSShort : () -> SizeOf(SShort)
val getUShort : LValue(UShort) -> Int
val setUShort : (LValue(UShort), Int) -> ()
val sizeOfUShort : () -> SizeOf(UShort)
val getSInt : LValue(SInt) -> Int
val setSInt : (LValue(SInt), Int) -> ()
val sizeOfSInt : () -> SizeOf(SInt)
val getUInt : LValue(UInt) -> Int
val setUInt : (LValue(UInt), Int) -> ()
val sizeOfUInt : () -> SizeOf(UInt)
val getSLong : LValue(SLong) -> Int
val setSLong : (LValue(SLong), Int) -> ()
val sizeOfSLong : () -> SizeOf(SLong)
val getULong : LValue(ULong) -> Int
val setULong : (LValue(ULong), Int) -> ()
val sizeOfULong : () -> SizeOf(ULong)
val getCFloat : LValue(CFloat) -> Float
val setCFloat : (LValue(CFloat), Float) -> ()
val sizeOfCFloat : () -> SizeOf(CFloat)
val getCDouble : LValue(CDouble) -> Double
val setCDouble : (LValue(CDouble), Double) -> ()
val sizeOfCDouble : () -> SizeOf(CDouble)

val addrOf : [t] LValue(t) -> CPtr(t)
val deref : [t] CPtr(t) -> LValue(t)
val addPtr : [t] (CPtr(t), SizeOf(t), Int) -> CPtr(t)
val getPtr : [t] (LValue(CPtr(t))) -> CPtr(t)
val setPtr : [t] (LValue(CPtr(t)), CPtr(t)) -> ()
val sizeOfPtr : [t] () -> SizeOf(CPtr(t))
val isNull : [t] CPtr(t) -> Bool
```

```
val sub : [t] (Arr(t), SizeOf(t), Int) -> LValue(t)
val toPtr : [t](Arr(t)) -> CPtr(t)
val toArr : [t](CPtr(t)) -> Arr(t)
val getArr : [t] (LValue(Arr(t))) -> Arr(t)
val setArr : [t] (LValue(Arr(t)), Arr(t)) -> ()

val getCFun : [t] (LValue(CFun(t))) -> CFun(t)
val setCFun : [t] (LValue(CFun(t)), CFun(t)) -> ()
val sizeOfCFun : [t] () -> SizeOf(CFun(t))
val isCFunNull : [t] CFun(t) -> Bool

val getCFun : [t] (LValue(CFun(t))) -> CFun(t)
val setCFun : [t] (LValue(CFun(t)), CFun(t)) -> ()
val sizeOfCFun : [t] () -> SizeOf(CFun(t))
val isCFunNull : [t] CFun(t) -> Bool

val copyString : [sign] CPtr(CChar(sign)) -> String
val copyBytes : (VoidPtr, Int) -> ByteArray
val viewAsBytes : (VoidPtr, Int) -> ByteArray
```

Bibliography

[FPR01] Fisher, K., R. Pucella, and J. Reppy. A framework for interoperability. In N. Benton and A. Kennedy (eds.), *Electronic Notes in Theoretical Computer Science*, vol. 59, 2001.