# Guile-GNOME: GObject

version 2.16.0, updated 12 June 2008

**Andy Wingo** (wingo at pobox.com)
**Martin Baulig** (baulig at suse.de)

This manual is for Guile-GNOME: GObject (version 2.16.0, updated 12 June 2008)

# Short Contents

# 1 (gnome-2)

## 1.1 Overview

Selects version 2 of the Guile-GNOME libraries. This module is used for its side effects; it exports no procedures.

## 1.2 Rationale

Early in the development of guile-gnome, we realized that at some point we might need to make incompatible changes. Of course, we would not want to force a correctly-written program to break when guile-gnome gets upgraded. For this reason, we decided to make guile-gnome parallel-installable. A program is completely specified when it indicates which version of guile-gnome it should use.

Guile-gnome has the concept of an API version, which indicates a stable API series. For example, a program written against API version 2 of guile-gnome will continue to work against all future releases of that API version. It is permitted to add interfaces within a stable series, but never to remove or change them incompatibly.

Changing the API version is expected to be a relatively infrequent operation. The current API version is 2.

There are two manners for a program to specify the guile-gnome version:

1. Via importing the `(gnome-`*`version`*`)` module.

   This special module alters guile's load path to include the path of the specified API version of guile-gnome. For example:

   ```
   (use-modules (gnome-2))
   ```

2. Via invoking guile as guile-gnome-*version*.

   This shell script is installed when building a particular version of guile-gnome, and serves to automatically load the `(gnome-`*`apiversion`*`)` module. For example, to get a repl ready for guile-gnome:

   ```
   $ guile-gnome-2
   ```

   To load a script with a particular version of guile-gnome:

   ```
   $ guile-gnome-2 -s script args...
   ```

   To specify the guile-gnome version in a script, you might begin the file with:

   ```
   #! /bin/sh
   # -*- scheme -*-
   exec guile-gnome-2 -s $0
   !#
   ;; scheme code here...
   ```

A program must select the guile-gnome version before importing any guile-gnome modules. Indeed, one cannot even import `(gnome gobject)` before doing so.

For a further rationale on parallel installability, see `http://ometer.com/parallel.html`.

## 1.3 Usage

# 2 (gnome gobject)

## 2.1 Overview

This is the Guile wrapper of `libgobject`, an implementation of a runtime, dynamic type system for C. Besides providing an object system to C, `libgobject`'s main design goal was to increase the ease with which C code can be wrapped by interpreted languages, such as Guile or Perl.

This module, `(gnome gobject)`, just re-exports procedures from other modules, so its documentation seems an opportune spot for a more tutorial-like introduction. So open up a Guile session and let's begin.

First, if you haven't done it, load the appropriate version of Guile-GNOME:

```
guile> (use-modules (gnome-2))
```

`(gnome gobject)` is based heavily on GOOPS, Guile's object system, so go ahead and load up that too:

```
guile> (use-modules (oop goops))
```

We will leave off the `guile>` prompt in the rest of this tutorial. When we want to show the value of an expression, we use ⇒:

```
(+ 3 5)
⇒ 8
```

## 2.2 Basic types

When communicating with `libgobject`, most values need to be strictly-typed. There is a type class corresponding to each basic type in C: `<gchar>`, `<guchar>`, `<gboolean>`, `<gint>`, `<guint>`, `<glong>`, `<gulong>`, `<gint64>`, `<guint64>`, `<gfloat>`, `<gdouble>`, and `<gchararray>`.

You can make instances of these class with `make`:

```
(make <gboolean> #:value #f)
⇒ #<gvalue <gboolean> 40529040 #f>

(make <guint> #:value 85)
⇒ #<gvalue <guint> 4054f040 85>

(make <gfloat> #:value 3.1415)
⇒ #<gvalue <gfloat> 40556af0 3.1414999961853>

(make <gchararray> #:value "Hello World!")
⇒ #<gvalue <gchararray> 4055af90 Hello World!>
```

You can get the normal Scheme values back with `gvalue->scm`:

```
(gvalue->scm (make <gchararray> #:value "Hello World!"))
⇒ "Hello World!"
```

## 2.3 Enums and flags

Enumerated values and bitflags are an essential part of many C APIs, and so they are specially wrapped in the GLib type system. You can create new enumerated types in Scheme by subclassing `<genum>`:

```
(define-class <foo> (<genum>)
  #:vtable '#((hello "Hello World" 1) (test "Test" 2)))
```

Instances are created with `make`, just like with the other types:

```
(make <foo> #:value 'hello)
(make <foo> #:value "Hello World")
(make <foo> #:value 1)

;; These three all do the same thing
⇒ #<gvalue <foo> 406275f8 (hello Hello World 1)>
```

If there is an already existing enum or flags class, you can get information about it:

```
(genum-class->value-table <foo>)
⇒ #((hello "Hello World" 1) (test "Test" 2))
```

Enums and flags have a special representation on the Scheme side. You can convert them to Scheme values as symbols, names, or as a numeric value.

```
(define foo (make <foo> #:value 'hello))
(genum->symbol foo)
⇒ hello
(genum->name foo)
⇒ "Hello World"
(genum->value foo)
⇒ 1
```

## 2.4 GType

All of the types that GLib knows about are available to Guile, regardless of which language defined them. GLib implements this via a type system, where every type has a name. So if you make a type called "Foo" in C, you can get to it in Scheme via `gtype-name->class`:

```
;; Retrieve the type for the foo enum we made earlier in the tutorial
(define copy-of-<foo> (gtype-name->class "Foo"))
(eq? <foo> copy-of-<foo>)
⇒ #t

(make copy-of-<foo> #:value 2)
```

```
⇒ #<gvalue <foo> 40535e50 (test Test 2)>
```

## 2.5 GObject

`<gobject>` (`GObject` in C) is the basic object type in `libgobject`. `(gnome gobject)` allows you to access existing GObject types, as well as to create new GObject types in Scheme.

Before we start, let's pull in some generic functions that reduce the amount of typing we have to do:

```
(use-modules (gnome gobject generics))
```

Let's assume we start with `<gtk-window>` from `(gnome gtk)`. The keyword arguments to `make` are interpreted as GObject properties to set:

```
(define window (make <gtk-window>
                 #:type 'toplevel #:title "Hello, World!"))
```

You can connect to signals on the new instance:

```
(connect window 'delete-event
         (lambda (window event)
           ;; Returns #t to ignore this event
           #t))

;; connect is a generic function implemented by
;; gtype-instance-signal-connect
```

And get and set properties...

```
(get window 'title)
⇒ "Hello, World!"
(set window 'resizable #f)

;; get and set are also generics, implemented by gobject-get-property
;; and gobject-set-property
```

## 2.6 Deriving your own GObject types

You can create new GObject types directly from Scheme, deriving either from a C object type or one you made in Scheme.

```
;; deriving from <gobject>
(define-class <test> (<gobject>)
  ;; a normal object slot
  my-data

  ;; an object slot exported as a gobject property
  (pub-data #:gparam (list <gparam-long> #:name 'test))
```

```
  ;; a signal with no arguments and no return value
  #:gsignal '(frobate #f))

;; deriving from <test> -- also inherits properties and signals
(define-class <hungry> (<test>))
```

Adding a signal automatically defines the default method:

```
;; This is the default handler for this signal.
(define-method (test:frobate (object <test>))
  (format #t "Frobating ~A\n" object))

;; We can override it for subclasses
(define-method (test:frobate (object <hungry>))
  (next-method) ;; chain up
  (format #t "I'm hungry\n"))

(emit (make <hungry>) 'frobate)
;; Try it!
```

You can override the `initialize`, `gobject:get-property`, and `gobject:set-property` methods. For an extended example, see `tic-tac-toe.scm` in the `gtk/examples/gtk` directory of the distribution.

## 2.7 Usage

# 3 (gnome gobject gtype)

## 3.1 Overview

Base support for the GLib type system.

The GLib runtime type system is broken into a number of modules, of which GType is the base. A GType is a simply a named type. Some types are fundamental and cannot be subclassed, such as integers. Others can form the root of complicated object hierarchies, such as `<gobject>`.

One can obtain the class for a type if you know its name. For example,

```
(gtype-name->class "guint64") ⇒ #<<gvalue-class> <guint64>>
```

A more detailed reference on the GLib type system may be had at `http://library.gnome.org/devel/gobject/stable/`.

## 3.2 Usage

`<gtype-class>` [Class]

The metaclass of all GType classes. Ensures that GType classes have a `gtype` slot, which records the primitive GType information for this class.

`<gtype-instance>` [Class]

The root class of all instantiatable GType classes. Adds a slot, `gtype-instance`, to instances, which holds a pointer to the C value.

`gtype-name->class` *name* [Primitive]

Return the `<gtype-class>` associated with the GType, *name*.

`class-name->gtype-name` *class-name* [Function]

Convert the name of a class into a suitable name for a GType. For example:

```
(class-name->gtype-name '<foo-bar>) ⇒ "FooBar"
```

`gruntime-error` *format-string . args* [Function]

Signal a runtime error. The error will be thrown to the key `gruntime-error`.

`gtype-instance-destroy!` *instance* [Primitive]

Release all references that the Scheme wrapper *instance* has on the underlying C value, and release pointers associated with the C value that point back to Scheme.

Normally, you don't need to call this function, because garbage collection will take care of resource management. However some `<gtype-class>` instances have semantics that require this function. The canonical example is that when a `<gtk-object>` emits the `destroy` signal, all code should drop their references to the object. This is, of course, handled internally in the `(gnome gtk)` module.

# 4 (gnome gobject gvalue)

## 4.1 Overview

GLib supports generic typed values via its GValue module. These values are wrapped in Scheme as instances of `<gvalue-class>` classes, such as `<gint>`, `<gfloat>`, etc.

In most cases, use of `<gvalue>` is transparent to the Scheme user. Values which can be represented directly as Scheme values are normally given to the user in their Scheme form, e.g. `#\a` instead of `#<gvalue <gchar> 3020c708 a>`. However, when dealing with low-level routines it is sometimes necessary to have values in `<gvalue>` form. The conversion between the two is performed via the `scm->gvalue` and `gvalue->scm` functions.

The other set of useful procedures exported by this module are those dealing with enumerated values and flags. These objects are normally represented on the C side with integers, but they have symbolic representations registered in the GLib type system.

On the Scheme side, enumerated and flags values are canonically expressed as `<gvalue>` objects. They can be converted to integers or symbols using the conversion procedures exported by this module. It is conventional for Scheme procedures that take enumerated values to accept any form for the values, which can be canonicalized using `(make <your-enum-type> #:value` *`value`*`)`, where *value* can be an integer, a symbol (or symbol list in the case of flags), or the string "nickname" (or string list) of the enumerated/flags value.

## 4.2 Usage

`<gvalue>` [Class]

`<gboolean>` [Class]

A `<gvalue>` class for boolean values.

`<gchar>` [Class]

A `<gvalue>` class for signed 8-bit values.

`<guchar>` [Class]

A `<gvalue>` class for unsigned 8-bit values.

`<gint>` [Class]

A `<gvalue>` class for signed 32-bit values.

`<guint>` [Class]

A `<gvalue>` class for unsigned 32-bit values.

`<glong>` [Class]

A `<gvalue>` class for signed "long" (32- or 64-bit) values.

`<gulong>` [Class]

A `<gvalue>` class for unsigned "long" (32- or 64-bit) values.

`<gint64>` [Class]

A `<gvalue>` class for signed 64-bit values.

`<guint64>` [Class]

A `<gvalue>` class for unsigned 64-bit values.

`<gfloat>` [Class]

A `<gvalue>` class for 32-bit floating-point values.

`<gdouble>` [Class]

A `<gvalue>` class for 64-bit floating-point values.

`<gchararray>` [Class]

A `<gvalue>` class for arrays of 8-bit values (C strings).

`<gboxed>` [Class]

A `<gvalue>` class for "boxed" types, a way of wrapping generic C structures. You won't see instances of this class, only of its subclasses.

`<gboxed-scm>` [Class]

A `<gboxed>` class for holding arbitrary Scheme objects.

`<gvalue-array>` [Class]

A `<gvalue>` class for arrays of `<gvalue>`.

`<gpointer>` [Class]

A `<gvalue>` class for opaque pointers.

`<genum>` [Class]

A `<gvalue>` base class for enumerated values. Users may define new enumerated value types via subclssing from `<genum>`, passing `#:vtable` *table* as an initarg, where *table* should be in a format suitable for passing to `genum-register-static`.

`<gflags>` [Class]

A `<gvalue>` base class for flag values. Users may define new flag value types via subclssing from `<gflags>`, passing `#:vtable` *table* as an initarg, where *table* should be in a format suitable for passing to `gflags-register-static`.

`genum-register-static` *name vtable* [Primitive]

Creates and registers a new enumerated type with name *name* with the C runtime. There must be no type with name *name* when this function is called.

The new type can be accessed by using `gtype-name->class`.

*vtable* is a vector describing the new enum type. Each vector element describes one enum element and must be a list of 3 elements: the element's nick name as a symbol, its name as a string, and its integer value.

```
(genum-register-static "Test"
  #((foo "Foo" 1) (bar "Bar" 2) (baz "Long name of baz" 4)))
```

`gflags-register-static` *name vtable* [Primitive]

Creates and registers a new flags `<gtype-class>` with name *name* with the C runtime.

The *vtable* should be in the format described in the documentation for `genum-register-static`.

**genum-class->value-table** *class* [Primitive]

Return a table of the values supported by the enumerated `<gtype-class>` *class*. The return value will be in the format described in `genum-register-static`.

**gflags-class->value-table** *class* [Primitive]

Return a table of the values supported by the flag `<gtype-class>` *class*. The return value will be in the format described in `gflags-register-static`.

**scm->gvalue** *class scm* [Primitive]

Convert a Scheme value into a `<gvalue>` of type *class*. If the conversion is not possible, raise a `gruntime-error`.

**gvalue->scm** *value* [Primitive]

Convert a `<gvalue>` into it normal scheme representation, for example unboxing characters into Scheme characters. Note that the Scheme form for some values is the `<gvalue>` form, for example with boxed or enumerated values.

**genum->symbol** *obj* [Function]

Convert the enumerated value *obj* from a `<gvalue>` to its symbol representation (its "nickname").

**genum->name** *obj* [Function]

Convert the enumerated value *obj* from a `<gvalue>` to its representation as a string (its "name").

**genum->value** *value* [Primitive]

Convert the enumerated value *obj* from a `<gvalue>` to its representation as an integer.

**gflags->value** *value* [Primitive]

Convert the flags value *obj* from a `<gvalue>` to its representation as an integer.

**gflags->symbol-list** *obj* [Function]

Convert the flags value *obj* from a `<gvalue>` to a list of the symbols that it represents.

**gflags->name-list** *obj* [Function]

Convert the flags value *obj* from a `<gvalue>` to a list of strings, the names of the values it represents.

**gflags->value-list** *obj* [Function]

Convert the flags value *obj* from a `<gvalue>` to a list of integers, which when `logand`'d together yield the flags' value.

# 5 (gnome gobject gparameter)

## 5.1 Overview

Parameters are constraints for values, both in type and in range. This module wraps the parameters code of the GLib type system, allowing parameters to be manipulated and created from Scheme.

There is a parameter class for each type of parameter: `<gparam-int>`, `<gparam-object>`, etc.

## 5.2 Usage

`<gparam>` [Class]

The base class for GLib parameter objects. (Doc slots)

`<gparam-char>` [Class]

Parameter for `<gchar>` values.

`<gparam-uchar>` [Class]

Parameter for `<guchar>` values.

`<gparam-boolean>` [Class]

Parameter for `<gboolean>` values.

`<gparam-int>` [Class]

Parameter for `<gint>` values.

`<gparam-uint>` [Class]

Parameter for `<guint>` values.

`<gparam-long>` [Class]

Parameter for `<glong>` values.

`<gparam-ulong>` [Class]

Parameter for `<gulong>` values.

`<gparam-int64>` [Class]

Parameter for `<gint64>` values.

`<gparam-uint64>` [Class]

Parameter for `<guint64>` values.

`<gparam-float>` [Class]

Parameter for `<gfloat>` values.

`<gparam-double>` [Class]

Parameter for `<gdouble>` values.

`<gparam-unichar>` [Class]

Parameter for Unicode codepoints, represented as `<guint>` values.

`<gparam-pointer>` [Class]

Parameter for `<gpointer>` values.

`<gparam-string>` [Class]

Parameter for `<gchararray>` values.

`<gparam-boxed>` [Class]

Parameter for `<gboxed>` values.

`<gparam-enum>` [Class]

Parameter for `<genum>` values.

`<gparam-flags>` [Class]

Parameter for `<gflags>` values.

`<gparam-spec-flags>` [Class]

A `<gflags>` type for the flags allowable on a `<gparam>`: `read`, `write`, `construct`, `construct-only`, and `lax-validation`.

`gparameter:uint-max` [Variable]

`gparameter:int-min` [Variable]

`gparameter:int-max` [Variable]

`gparameter:ulong-max` [Variable]

`gparameter:long-min` [Variable]

`gparameter:long-max` [Variable]

`gparameter:uint64-max` [Variable]

`gparameter:int64-min` [Variable]

`gparameter:int64-max` [Variable]

`gparameter:float-max` [Variable]

`gparameter:float-min` [Variable]

`gparameter:double-max` [Variable]

`gparameter:double-min` [Variable]

`gparameter:byte-order` [Variable]

# 6 (gnome gobject gclosure)

## 6.1 Overview

The GLib type system supports the creation and invocation of "closures", objects which can be invoked like procedures. Its infrastructure allows one to pass a Scheme function to C, and have C call into Scheme, and vice versa. In Scheme, `<gclosure>` holds a Scheme procedure, the `<gtype>` of its return value, and a list of the `<gtype>`'s of its arguments. Closures can be invoked with `gclosure-invoke`.

However since on the C level, closures do not carry a description of their argument and return types, when we invoke a closure we have to be very explicit about the types involved. For example:

```
(gclosure-invoke (make <gclosure>
                  #:return-type <gint>
                  #:param-types (list <gulong>)
                  #:func (lambda (x) (* x x)))
                 <gulong>
                 (scm->gvalue <gulong> 10))
⇒ 100
```

## 6.2 Usage

`<gclosure>` [Class]

The Scheme representation of a GLib closure: a typed procedure object that can be passed to other languages.

`gclosure-invoke` *closure return_type args* [Primitive]

Invoke a closure.

A `<gclosure>` in GLib's abstraction for a callable object. This abstraction carries no type information, so the caller must supply all arguments as typed <gvalue> instances, which may be obtained by the scheme procedure, `scm->gvalue`.

As you can see, this is a low-level function. In fact, it is not used internally by the `guile-gobject` bindings.

# 7 (gnome gobject gsignal)

## 7.1 Overview

GSignal is a mechanism by which code, normally written in C, may expose extension points to which closures can be connected, much like Guile's hooks. Instantiatable types can have signals associated with them; for example, `<gtk-widget>` has an `expose` signal that will be "fired" at certain well-documented points.

Signals are typed. They specify the types of their return value, and the types of their arguments.

This module defines routines for introspecting, emitting, connecting to, disconnecting from, blocking, and unblocking signals. Additionally it defines routines to define new signal types on instantiatable types.

## 7.2 Usage

`<gsignal>` [Class]

A `<gsignal>` describes a signal on a `<gtype-instance>`: its name, and how it should be called.

`gtype-class-get-signals` *class tail* [Primitive]

Returns a list of signals belonging to *class* and all parent types.

`gtype-class-get-signal-names` *class* [Function]

Returns a vector of signal names belonging to *class* and all parent classes.

`gtype-instance-signal-emit` *object name args* [Primitive]

`gtype-instance-signal-connect` *object name func . after?* [Function]

Connects *func* as handler for the `<gtype-instance>` *object*'s signal *name*.

*name* should be a symbol. *after* is boolean specifying whether the handler is run before (`#f`) or after (`#t`) the signal's default handler.

Returns an integer number which can be used as arugment of `gsignal-handler-block`, `gsignal-handler-unblock`, `gsignal-handler-disconnect` and `gsignal-handler-connected?`.

`gtype-instance-signal-connect-after` *object name func* [Function]

Convenience function for calling `gtype-instance-signal-connect` with *after* = `#t`.

`gsignal-handler-block` *instance handler_id* [Primitive]

`gsignal-handler-unblock` *instance handler_id* [Primitive]

`gsignal-handler-disconnect` *instance handler_id* [Primitive]

`gsignal-handler-connected?` *instance handler_id* [Primitive]

`gtype-class-create-signal` *class name return-type param-types* [Function]

Create a new signal associated with the `<gtype-class>` *class*.

*name* should be a symbol, the name of the signal. *return-type* should be a `<gtype-class>` object. *param-types* should be a list of `<gtype-class>` objects.

In a bit of an odd interface, this function will return a new generic function, which will be run as the signal's default handler, whose default method will silently return an unspecified value. The user may define new methods on this generic to provide alternative default handler implementations.

# 8 (gnome gobject gobject)

## 8.1 Overview

GObject is what is commonly understood as *the* object system for GLib. This is not strictly true. GObject is *one* implementation of an object system, built on the other modules: GType, GValue, GParameter, GClosure, and GSignal.

Similarly, this Guile module provides integration with the GObject object system, built on the Guile modules that support GType, GValue, GParameter, GClosure, and GSignal.

The main class exported by this module is `<gobject>`. `<gobject>` classes can be subclassed by the user, which will register new subtypes with the GType runtime type system. `<gobject>` classes are are also created as needed when wrapping GObjects that come from C, for example from a function's return value.

Besides supporting derivation, and signals like other `<gtype-instance>` implementations, `<gobject>` has the concept of *properties*, which are `<gvalue>`'s associated with the object. The values are constrained by `<gparam>`'s, which are associated with the object's class. This module exports the necessary routines to query, get, and set `<gobject>` properties.

In addition, this module defines the `<ginterface>` base class, whose subclasses may be present as mixins of `<gobject>` classes. For example:

```
(use-modules (gnome gtk) (oop goops))
(class-direct-supers <gtk-widget>) ⇒
   (#<<gobject-class> <atk-implementor-iface> 3033bad0>
    #<<gobject-class> <gtk-object> 3034bc90>)
```

In this example, we see that `<gtk-widget>` has two superclasses, `<gtk-object>` and `<atk-implementor-iface>`. The second is an interface implemented by the `<gtk-widget>` class. See `gtype-interfaces` for more details.

## 8.2 Usage

`<gobject>` [Class]

The base class for GLib's default object system.

`<gobject>`'s metaclass understands a new slot option, `#:gparam`, which will export a slot as a `<gobject>` property. The default implementation will set and access the value from the slot, but you can customize this by writing your own methods for `gobject:set-property` and `gobject:get-property`.

In addition, the metaclass also understands `#:gsignal` arguments, which define signals on the class, and define the generics for the default signal handler. See `gtype-class-define-signal` for more information.

For example:

```
;; deriving from <gobject>
(define-class <test> (<gobject>)
 ;; a normal object slot
```

```
          my-data

          ;; an object slot exported as a gobject property
          (pub-data #:gparam (list <gparam-long> #:name 'test))

          ;; likewise, using non-default parameter settings
          (foo-data #:gparam (list <gparam-long> #:name 'foo
                                   #:minimum -3 #:maximum 1000
                                   #:default-value 42))

          ;; a signal with no arguments and no return value
          #:gsignal '(frobate #f)

          ;; a signal with arguments and a return value
          #:gsignal (list 'frobate <gboolean> <gint> <glong>))

        ;; deriving from <test> -- also inherits properties and signals
        (define-class <hungry> (<test>))
```

`<gobject>` classes also expose a slot for each GObject property defined on the class, if such a slot is not already defined.

`<ginterface>` [Class]

The base class for GLib's interface types. Not derivable in Scheme.

`<gparam-object>` [Class]

Parameter for `<gobject>` values.

`gtype-register-static` *name parent_class* [Primitive]

Derive a new type named *name* from *parent_class*. Returns the new `<gtype-class>`. This function is called when deriving from `<gobject>`; users do not normally call this function directly.

`gobject:get-property` [Generic]

Called to get a gobject property. Only properties directly belonging to the object's class will come through this function; superclasses handle their own properties.

Takes two arguments: the object and the property name.

Call `(next-method)` in your methods to invoke the default handler

`gobject:get-property (`*`object`* `<gobject>) (`*`name`* `<symbol>)` [Method]

The default implementation of `gobject:get-property`, which calls `(slot-ref obj name)`.

`gobject:set-property` [Generic]

Called to set a gobject property. Only properties directly belonging to the object's class will come through this function; superclasses handle their own properties.

Takes three arguments: the object, the property name, and the value.

Call `(next-method)` in your methods to invoke the default handler.

`gobject:set-property` (*object* `<gobject>`) (*name* `<symbol>`) (*value* `<top>`) [Method]

The default implementation of `gobject:set-property`, which sets slots on the object.

`gobject-class-get-properties` *class* [Primitive]

`gobject-class-find-property` *class name* [Function]

Returns a property named *name* (a symbol), belonging to *class* or one of its parent classes, or `#f` if not found.

`gobject-class-get-property-names` *class* [Primitive]

`gobject-get-property` *object name* [Primitive]

Gets a the property named *name* (a symbol) from *object*.

`gobject-set-property` *object name value* [Primitive]

Sets the property named *name* (a symbol) on *object* to *init-value*.

# 9 (gnome gobject generics)

## 9.1 Overview

Generic functions for procedures in the `(gnome gobject)` module.

### 9.1.1 Mapping class libraries to Scheme

Guile-GNOME exists to wrap a C library, `libgobject`, its types, and the set of libraries that based themselves on the GLib types.

Procedure invocation feels very similar in Scheme and in C. For example, the C `gtk_widget_show (widget)` transliterates almost exactly to the Scheme `(gtk-widget-show widget)`.

GLib-based libraries are not random collections of functions, however. GLib-based libraries also implement classes and methods, insofar that it is possible in C. For example, in the above example, `show` may be seen to be a method on instances of the `<gtk-widget>` class.

Indeed, other object-oriented languages such as Python express this pattern directly, translating the `show` operation as the pleasantly brief `widget.show()`. However this representation of methods as being bound to instances, while common, has a number of drawbacks.

The largest drawback is that the method itself is not bound to a generic operation. For example, mapping the `show` operation across a set of widgets cannot be done with the straightforward `map(show, set)`, because there is no object for the `show` operation. Instead the user must locally bind each widget to a variable in order to access a method of the abstract `show` operation: `map(lambda widget: widget.show(), set)`.

Additionally, most languages which express methods as bound to instances only select the method via the type of the first (implicit) argument. The rule for these lanugages is, "`gtk-widget-show` is an applicable method of the `show` operation when the first argument to `show` is a `<gtk-widget>`." Note the lack of specification for other arguments; the same object cannot have two applicable methods of the `show` operation. A more complete specification would be, "`gtk-widget-show` is an applicable method of the `show` operation when applied to one argument, a `<gtk-widget>`." It is a fine difference, but sometimes important.

For these and other reasons, the conventional way to implement generic operations in Lisp has been to define *generic functions*, and then associate specific methods with those functions. For example, one would write the following:

```
;; defining a generic function, and one method implementation
(define-generic show)
(define-method (show (widget <gtk-widget>))
  (gtk-widget-show widget))

;; invoking the generic function
(show my-widget)
```

One benefit of this approach is that method definitions can be made far away in space and time from type definitions. This leads to a more dynamic environment, in which methods can be added to existing types at runtime, which then can apply to existing instances.

### 9.1.2 The semantics of generic functions in Guile-GNOME

Naturally, there is an impedance mismatch between the conventions used in the C libraries and their Scheme equivalents. Operations in GLib-based libraries do not form a coherent whole, in the sense that there is no place that defines the meaning of an abstract `show` operation. For example, `gtk-widget-set-state`, which can make a widget become uneditable, and `gst-element-set-state`, which can start a video player, would both map to the generic function `set-state`, even though they have nothing to do with each other besides their name.

There is no conflict here; the methods apply on disjoint types. However there is a problem of modularity, in that *both methods must be defined on the same generic function*, so that `(set-state foo bar)` picks the correct method, depending on the types of *foo* and *bar*.

This point leads to the conclusion that *generic functions in Guile-GNOME have no abstract meaning, apart from their names.* Semantically, generics in Guile-GNOME are abbreviations to save typing, not abstract operations with defined meanings.

### 9.1.3 Practicalities

This module defines a number of "abbreviations", in the form of generic functions, for operations on types defined in the `(gnome gobject)` modules. Generic functions for generated bindings like `(gnome gtk)` are defined in another module, `(gnome gw generics)`, which re-exports the public bindings from this module.

## 9.2 Usage

`get` [Generic]

`get` (*object* <gobject>) (*name* <symbol>) [Method]

A shorthand for `gobject-get-property`.

`set` [Generic]

`set` (*object* <gobject>) (*name* <symbol>) (*value* <top>) [Method]

A shorthand for `gobject-set-property`.

`emit` [Generic]

`emit` (*object* <gtype-instance>) (*name* <symbol>) (*args* <top>)... [Method]

A shorthand for `gtype-instance-signal-emit`.

`connect` [Generic]

`connect` (*object* <gtype-instance>) (*name* <symbol>) (*func* <procedure>) [Method]

A shorthand for `gtype-instance-signal-connect`.

`connect` (*args* <top>)... [Method]
The core Guile implementation of the connect(2) POSIX call

`connect-after` [Generic]

`connect-after` (*object* <gtype-instance>) (*name* <symbol>) (*func* <procedure>) [Method]
A shorthand for `gtype-instance-signal-connect-after`.

`block` [Generic]

`block` (*object* <gtype-instance>) (*id* <top>) [Method]
A shorthand for `gsignal-handler-block`.

`unblock` [Generic]

`unblock` (*object* <gtype-instance>) (*id* <top>) [Method]
A shorthand for `gsignal-handler-unblock`.

`disconnect` [Generic]

`disconnect` (*object* <gtype-instance>) (*id* <top>) [Method]
A shorthand for `gsignal-handler-disconnect`.

`connected?` [Generic]

`connected?` (*object* <gtype-instance>) (*id* <top>) [Method]
A shorthand for `gsignal-handler-connected?`.

`invoke` [Generic]

`invoke` (*closure* <gclosure>) (*args* <top>)... [Method]
A shorthand for `gclosure-invoke`.

`create-signal` [Generic]

`create-signal` (*class* <gtype-class>) (*name* <symbol>) (*return-type* <top>) (*param-types* <top>) [Method]
A shorthand for `gtype-class-create-signal`.

`get-signals` [Generic]

`get-signals` (*class* <gtype-class>) [Method]
A shorthand for `gtype-class-get-signals`.

`get-properties` [Generic]

`get-properties` (*class* <gtype-class>) [Method]
A shorthand for `gobject-class-get-properties`.

`get-property-names` [Generic]

`get-property-names` (*class* <gtype-class>) [Method]
A shorthand for `gobject-class-get-property-names`.

`find-property` [Generic]

`find-property` (*class* <gtype-class>) (*name* <symbol>) [Method]
A shorthand for `gobject-class-find-property`.

# 10 (gnome gobject utils)

## 10.1 Overview

Common utility routines.

## 10.2 Usage

`GStudlyCapsExpand` *nstr* [Function]

Expand the StudlyCaps *nstr* to a more schemey-form, according to the conventions of GLib libraries. For example:

```
(GStudlyCapsExpand "GSource") ⇒ g-source
(GStudlyCapsExpand "GtkIMContext") ⇒ gtk-im-context
(GStudlyCapsExpand "GtkHBox") ⇒ gtk-hbox
```

`gtype-name->scheme-name-alist` [Variable]

An alist of exceptions to the name transformation algorithm implemented in `GStudlyCapsExpand`.

`gtype-name->scheme-name` *type-name* [Function]

Transform a name of a `<gtype>`, such as "GtkWindow", to a scheme form, such as `gtk-window`, taking into account the exceptions in `gtype-name->scheme-name-alist`, and trimming trailing dashes if any.

`gtype-name->class-name` *type-name* [Function]

Transform a name of a `<gtype>`, such as "GtkWindow", to a suitable name of a Scheme class, such as `<gtk-window>`. Uses `gtype-name->scheme-name`.

`gtype-class-name->method-name` *class-name name* [Function]

Generate the name of a method given the name of a `<gtype>` and the name of the operation. For example:

```
(gtype-name->method-name "GtkFoo" "bar") ⇒ gtk-foo:bar
```

Uses `gtype-name->scheme-name`.

`re-export-modules` *. args* [Special Form]

Re-export the public interface of a module or modules. Invoked as `(re-export-modules (mod1) (mod2)...)`.

`define-macro-with-docs` *form docs . body* [Special Form]

`define-with-docs` *name docs val* [Special Form]

Define *name* as *val*, documenting the value with *docs*.

`define-generic-with-docs` *name documentation* [Special Form]

Define a generic named *name*, with documentation *documentation*.

`define-class-with-docs` *name supers docs . rest* [Special Form]

Define a class named *name*, with superclasses *supers*, with documentation *docs*.

`unless` *test . body* [Special Form]

`with-accessors` *names . body* [Special Form]

# 11 (gnome gw generics)

## 11.1 Overview

This module exists so that all `(gnome gw)` modules have a common place to put their generic functions. Whenever a wrapset is loaded, it adds method definitions to generics defined in this module.

See the documentation for `(gnome gobject generics)` for more notes about generic functions in Guile-GNOME. This module re-exports bindings from `(gnome gobject generics)`, so there is no need to import them both.

## 11.2 Usage

# 12 (gnome gw support gobject)

## 12.1 Overview

G-Wrap support for `(gnome gobject)` types. Code in this module is only loaded when generating wrapsets; as such, it is not for end users.

## 12.2 Usage

`<gobject-wrapset-base>` [Class]

The base class for G-Wrap wrapsets that use `<gobject>` types.

`add-type-alias!` [Generic]

`add-type-alias!` (*wrapset* `<gobject-wrapset-base>`) (*alias* `<string>`) (*name* `<symbol>`) [Method]

Add a type alias to *wrapset*, that the string *alias* is associated with the type named *symbol*. For example, `"GtkWindow*"` might be associated with a type named `<gtk-window>`. See `lookup-type-by-alias`.

`lookup-type-by-alias` [Generic]

`lookup-type-by-alias` (*wrapset* `<gobject-wrapset-base>`) (*name* `<string>`) [Method]

Lookup a type aliased *name* in *wrapset*, and all wrapsets on which *wrapset* depends. This interface is used by `load-defs` to associate G-Wrap types with the strings parsed out of the C header files.

`add-type-rule!` [Generic]

`add-type-rule!` (*self* `<gobject-wrapset-base>`) (*param-type* `<string>`) (*typespec* `<top>`) [Method]

Add a type rule to *wrapset*, that the string *param-type* maps directly to the g-wrap typespec *typespec*. For example, `"int*"` might map to the typespec `(int out)`. See `find-type-rule`.

`find-type-rule` [Generic]

`find-type-rule` (*self* `<gobject-wrapset-base>`) (*param-type* `<string>`) [Method]

See if the parameter type *param-type* has a type rule present in *wrapset* or in any wrapset on which *wrapset* depends. This interface is used by `load-defs` to associate G-Wrap typespecs with the strings parsed out of the C header files.

`<gobject-type-base>` [Class]

A base G-Wrap type class for GLib types.

`<gobject-classed-type>` [Class]

A base G-Wrap type class for classed GLib types (see `gtype-classed?`).

`gtype-id` [Generic]

`gtype-id` (*o* `<gobject-custom-gvalue-type>`) [Method]

`gtype-id` (*o* `<gobject-custom-boxed-type>`) [Method]

`gtype-id` (*o* `<gobject-class-type>`) [Method]

`gtype-id` (*o* `<gobject-flags-type>`) [Method]

`gtype-id` (*o* `<gobject-enum-type>`) [Method]

`gtype-id` (*o* `<gobject-interface-type>`) [Method]

`gtype-id` (*o* `<gobject-pointer-type>`) [Method]

`gtype-id` (*o* `<gobject-boxed-type>`) [Method]

`gtype-id` (*o* `<gobject-instance-type>`) [Method]

`gtype-id` (*o* `<gobject-classed-pointer-type>`) [Method]

`gtype-id` (*o* `<gobject-classed-type>`) [Method]

`<gobject-classed-pointer-type>` [Class]

A base G-Wrap type class for for classed GLib types whose values are pointers.

`unwrap-null-checked` [Generic]

`unwrap-null-checked` (*value* `<gw-value>`) (*status-var* `<top>`) (*code* `<top>`) [Method]

Unwrap a value into a C pointer, optionally unwrapping `#f` as `NULL`.

This function checks the typespec options on *value*, which should be a `<gw-value>`. If the `null-ok` option is set (which is only the case for value classes with `null-ok` in its `#:allowed-options`), this function generates code that unwraps `#f` as `NULL`. If `null-ok` is unset, or the value is not `#f`, *code* is run instead.

`wrap-instance!` [Generic]

`wrap-instance!` (*ws* `<gobject-wrapset-base>`) (*args* `<top>`)... [Method]

Define a wrapper for a specific instantiatable (`<gtype-instance>`-derived) type in *ws*. Required keyword arguments are `#:ctype` and `#:gtype-id`. For example,

```
(wrap-instance! ws #:ctype "GtkWidget"
                   #:gtype-id "GTK_TYPE_WIDGET")
```

Normally only called from `load-defs`.

`wrap-boxed!` [Generic]

`wrap-boxed!` (*ws* `<gobject-wrapset-base>`) (*args* `<top>`)... [Method]

Define a wrapper for a specific boxed type in *ws*. Required keyword arguments are `#:ctype` and `#:gtype-id`, as in `wrap-instance!`.

`wrap-pointer!` [Generic]

`wrap-pointer!` (*ws* `<gobject-wrapset-base>`) (*args* `<top>`)... [Method]

Define a wrapper for a specific pointer type in *ws*. Required keyword arguments are `#:ctype` and `#:gtype-id`, as in `wrap-instance!`.

`wrap-opaque-pointer!` *ws ctype* [Function]

Define a wrapper for an opaque pointer with the C type *ctype*. It will not be possible to create these types from Scheme, but they can be received from a library, and passed as arguments to other calls into the library.

`wrap-freeable-pointer!` *ws ctype free* [Function]

foo

`wrap-refcounted-pointer!` *ws ctype ref unref* [Function]

foo

`wrap-structure!` *ws ctype wrap unwrap* [Function]

Define a wrapper for structure values of type *ctype*.

*wrap* and *unwrap* are the names of C functions to convert a C structure to Scheme and vice versa, respectively. When in a function call, parameters of this type of the form '*StructName**' are interpreted as 'out' parameters, while 'const-*StructName**' are treated as 'in' parameters.

Note that *ctype* should be the type of the structure, not a pointer to the structure.

`wrap-interface!` [Generic]

`wrap-interface!` (*ws* `<gobject-wrapset-base>`) (*args* `<top>`)... [Method]

Define a wrapper for an interface type in *ws*. Required keyword arguments are `#:ctype` and `#:gtype-id`, as in `wrap-instance!`.

`wrap-flags!` [Generic]

`wrap-flags!` (*ws* `<gobject-wrapset-base>`) (*args* `<top>`)... [Method]

Define a wrapper for a flags type in *ws*. Required keyword arguments are `#:ctype` and `#:gtype-id` or `#:values`, as in `wrap-enum!`.

`wrap-gobject-class!` [Generic]

`wrap-gobject-class!` (*ws* `<gobject-wrapset-base>`) (*args* `<top>`)... [Method]

Define a wrapper for GObject class values *ws*. Required keyword arguments are `#:ctype` and `#:gtype-id`, as in `wrap-instance!`.

`#:ctype` should refer to the type of the class and not the instance; e.g. `"GtkWidgetClass"` and not `"GtkWidget"`. This function will not be called by `load-defs`, and should be invoked manually in a wrapset as needed.

`wrap-custom-boxed!` *ctype gtype wrap unwrap* [Special Form]

Wrap a boxed type using custom wrappers and unwrappers.

FIXME: missing a wrapset argument!

*ctype* and *gtype* are as `#:ctype` and `#:gtype-id` in `wrap-instance!`. *wrap* and *unwrap* are G-Wrap forms in which `scm-var` and `c-var` will be bound to the names of the SCM and C values, respectively. For example:

```
(wrap-custom-boxed!
 "GdkRectangle" "GDK_TYPE_RECTANGLE"
 (list scm-var " = "
       c-var " ?  scm_gdk_rectangle_to_scm (" c-var ")"
       " : SCM_BOOL_F;")
 (list c-var " = scm_scm_to_gdk_rectangle (" scm-var ");"))
```

**wrap-custom-gvalue!** *ctype gtype wrap-func unwrap-func* [Special Form]

Wrap a GValue type using custom wrap and unwrap functions.

FIXME: missing a wrapset argument!

*ctype* and *gtype* are as `#:ctype` and `#:gtype-id` in `wrap-instance!`. *wrap-func* and *unwrap-func* are names of functions to convert to and from Scheme values, respectively. For example:

```
(wrap-custom-gvalue! "GstFraction" "GST_TYPE_FRACTION"
                     "scm_from_gst_fraction"
                     "scm_to_gst_fraction")
```

# 13 (gnome gw support defs)

## 13.1 Overview

This module serves as a way to automatically populate G-Wrap wrapsets using information parsed out of C header files.

First, the C header files are parsed into S-expression API description forms and written into `.defs` files. These files are typically included in the distribution, and regenerated infrequently. Then, the binding author includes a call to `load-defs` in their G-Wrap wrapset definition, which loads those API definitions into the wrapset.

The `.defs` files are usually produced using the API scanner script, `h2defs.py`, included in the Guile-GNOME source distribution.

Code in this module is only loaded when generating wrapsets; as such, it is not for end users.

As an example, ATK is wrapped with the following code, from `atk/gnome/gw/atk-spec.scm`:

```
(define-module (gnome gw atk-spec)
  #:use-module (oop goops)
  #:use-module (gnome gw support g-wrap)
  #:use-module (gnome gw gobject-spec)
  #:use-module (gnome gw support gobject)
  #:use-module (gnome gw support defs))

(define-class <atk-wrapset> (<gobject-wrapset-base>)
  #:id 'gnome-atk
  #:dependencies '(standard gnome-glib gnome-gobject))

(define-method (global-declarations-cg (self <atk-wrapset>))
  (list
   (next-method)
   "#include <atk/atk.h>\n"
   "#include <atk/atk-enum-types.h>\n"))

(define-method (initialize (ws <atk-wrapset>) initargs)
  (next-method ws (append '(#:module (gnome gw atk)) initargs))
  ;; manually wrap AtkState as a 64 bit uint
  (add-type-alias! ws "AtkState" 'unsigned-int64)
  (load-defs-with-overrides ws "gnome/defs/atk.defs"))
```

The wrapper-specifiction modules are actually installed, along with the .defs files, so that other wrappers which use ATK's types, such as GTK+, can have them available.

A full discussion of the Makefile mechanics of how to generate and compile the C file, or how to interact with the wrapset objects, is probably prone to bitrot here. Your best bet is to poke at Guile-GNOME's source, or especially the source of a module distributed independently of `guile-gnome-platform`, such as `guile-gnome-libwnck`.

Further details about the procedural API available for use e.g. within the wrapset's `initialize` function can be found in the documentation for `(gnome gw support gobject)`, and in G-Wrap's documentation.

## 13.2 Usage

`load-defs` *ws file* [*overrides* = #f] [Function]

Load G-Wrap type and function information from *file* into the G-Wrap wrapset *ws*.

*file* should be a relative path, which will be searched in the vicinity of Guile's `%load-path`. `include` directives in the file will be searched relative to the absolute path of the file.

The following forms are understood: `define-enum`, `define-flags`, `define-object`, `define-interface`, `define-pointer`, `define-boxed`, `define-function`, `define-method`, `ignore`, `ignore-glob`, and `ignore-types`.

The optional argument, *overrides*, specifies the location of an overrides file that will be spliced into the `.defs` file at the point of an `(include overrides)` form.

`load-defs-with-overrides` *ws defs* [Function]

Equivalent to:

```
(load-defs ws defs
           (string-append "gnome/overrides/"
                          (basename defs)))
```

# 14 (gnome gw support gtk-doc)

## 14.1 Overview

This module exports two high-level procedures to transform the Docbook files generated by GTK-Doc into texinfo.

GTK-Doc (`http://www.gtk.org/gtk-doc/`) is commonly used to document GObject-based libraries, such as those that Guile-GNOME wraps. In a typical build setup, GTK-Doc generates a reference manual with one XML file per section. The routines in this module attempt to recreate those sections, but in Texinfo instead of Docbook, and which document the Scheme modules instead of the upstream C libraries.

The tricky part of translating GTK-Doc's documentation is not the vocabulary (Docbook), but that it documents C functions which have different calling conventions than Scheme. For example, a C function might take four `double*` arguments, but in Scheme the function would return four rational values. Given only the C prototype, the code in this module will make an attempt to determine what the Scheme function's arguments will be based on some heuristics.

In most cases, however, we can do better than heuristics, because we have the G-Wrap information that describes the relationship between the C function and the Scheme wrapper. In that way we can know exactly what the input and output arguments are for a particular function.

The `gtk-doc->texi-stubs` function is straightforward. It extracts the "header" in a set of GTK-Doc files, translates them into texinfo, writing them out one by one to files named '`section-`*`foo`*`.texi`', where *foo* is the name of the XML file. It is unclear whether it is best to continously generate these sections when updating the manuals, or whether this "stub" generation should be run only once when the documentation is initially generated, and thereafter maintained by hand. Your call!

`gtk-doc->texi-defuns` is slightly more complicated, because you have the choice as to whether to use heuristics or the g-wrap method for determining the arguments. See its documentation for more information.

Both of these functions are designed to be directly callable from the shell. Here is a makefile snippet suitable for using the heuristics method for defuns generation:

```
GTK_DOC_TO_TEXI_STUBS = \
  '((@ (gnome gw support gtk-doc) gtk-doc->texi-stubs) \
  (cdr (program-arguments)))'
GTK_DOC_DEFUN_METHOD = heuristics
GTK_DOC_DEFUN_ARGS = (your-module-here)
GTK_DOC_TO_TEXI_DEFUNS = "(apply (@ (gnome gw support gtk-doc) \
   gtk-doc->texi-defuns) (cadr (program-arguments)) \
   '$(GTK_DOC_DEFUN_METHOD) '($(GTK_DOC_DEFUN_ARGS)) \
   (cddr (program-arguments)))"
GUILE = $(top_builddir)/dev-environ guile

generate-stubs:
	$(GUILE) $(GUILE_FLAGS) -c $(GTK_DOC_TO_TEXI_STUBS) \
```

```
            $(docbook_xml_files)

    generate-defuns:
     $(GUILE) $(GUILE_FLAGS) -c $(GTK_DOC_TO_TEXI_DEFUNS) \
            ./overrides.texi $(docbook_xml_files)
```

To make the above snippet work, you will have to define `$(docbook_xml_files)` as the set of docbook XML files to transform. To use the G-Wrap method, try the following:

```
wrapset_module = (gnome gw $(wrapset_stem)-spec)
wrapset_name = gnome-$(wrapset_stem)
GTK_DOC_DEFUN_METHOD = g-wrap
GTK_DOC_DEFUN_ARGS = $(wrapset_module) $(wrapset_name)
```

Set `$(wrapset_stem)` to the stem of the wrapset name, e.g. `pango`, and there you are.

## 14.2 Usage

`gtk-doc->texi-stubs` *files* [Function]

Generate a section overview texinfo file for each docbook XML file in *files*.

The files will be created in the current directory, as described in the documentation for `(gnome gw support gtk-doc)`. They will include a file named `defuns-`*`file`*`.texi`, which should probably be created using `gtk-doc->texi-defuns`.

`gtk-doc->texi-defuns` *overrides method args . files* [Function]

Generate documentation for the types and functions defined in a set of docbook files genearted by GTK-Doc.

*overrides* should be a path to a texinfo file from which `@deffn` overrides will be taken. *method* should be either `g-wrap` or `heuristics`, as discussed in the `(gnome gw support gtk-doc)` documentation. *files* is the list of docbook XML files from which to pull function documentation.

*args* should be a list, whose form depends on the *method*. For `g-wrap`, it should be two elements, the first the name of a module that, when loaded, will load the necessary wrapset into the g-wrap runtime. For example, `(gnome gw glib-spec)`. The second argument should be the name of the wrapset, e.g. `gnome-glib`.

If *method* is `heuristics`, *args* should have only one element, the name of the module to load to check the existence of procedures, e.g. `(cairo)`.

`check-documentation-coverage` *modules texi* [Function]

Check the coverage of generated documentation.

*modules* is a list of module names, and *texi* is a path to a texinfo file. The set of exports of *modules* is checked against the set of procedures defined in *texi*, resulting in a calculation of documentation coverage, and the output of any missing documentation to the current output port.

`generate-undocumented-texi` *modules texi* [Function]

Verify the bindings exported by *modules* against the documentation in *texi*, writing documentation for any undocumented symbol to `undocumented.texi`.

*modules* is a list of module names, and *texi* is a path to a texinfo file.

# 15 (gnome gw support modules)

## 15.1 Overview

Support routines for automatically-generated scheme G-Wrap modules.

## 15.2 Usage

`export-all-lazy!` *symbols* [Function]

Export the *symbols* from the current module.

Most generic functions and classes that G-Wrap defines are bound lazily, as needed in evaluation. This is done by placing module binder procedures on the generated modules. However, if we export all symbols by name, this will force the binding eagerly for all values, which is slow.

This procedure exports all bindings named in *symbols* that are already bound in the current module, and then installs a module binder procedure on the public interface, which allows lazy binding to work.

`re-export-modules` . *args* [Special Form]

Re-export the public interface of a module; used like `use-modules`.

# Type Index

# Function Index

## W