WARNING: (srfi srfi-35): ‘every’ imported from both (oop goops util) and (srfi srfi-1) WARNING: (srfi srfi-35): ‘any’ imported from both (oop goops util) and (srfi srfi-1) WARNING: (g-wrap util): imported module (srfi srfi-34) overrides core binding ‘raise’ WARNING: (g-wrap): imported module (srfi srfi-34) overrides core binding ‘raise’ WARNING: (g-wrap rti): imported module (srfi srfi-34) overrides core binding ‘raise’ WARNING: (g-wrap c-types): imported module (srfi srfi-34) overrides core binding ‘raise’ WARNING: (gnome gw support g-wrap): ‘declarations-cg’ imported from both (g-wrap c-codegen) and (g-wrap scm-codegen) WARNING: (gnome gw support defs): imported module (srfi srfi-34) overrides core binding ‘raise’ WARNING: (gnome gw support gtk-doc): imported module (sxml xpath) overrides core binding ‘filter’

Guile-GNOME: GObject

version 2.15.93, updated 25 August 2007

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

This manual is for Guile-GNOME: GObject (version 2.15.93, updated 25 August 2007)

Short Contents

1 (gnome gobject gtype)

1.1 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 named type that has a number of properties. 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 `<gtype>` object for a type if you know its name. For example,

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

`<gtype>` objects are low-level constructs. In Scheme, it is more usual to work with GOOPS type objects. Each `<gtype>` corresponds to one GOOPS object, which may be obtained programmatically using `gtype->class`.

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

1.2 Usage

`gtype?` *type* [Primitive]
Returns `#t` if *type* is a GType and `#f` if not.

`gtype-is-a?` *type is_a_type* [Primitive]
Returns `#t` if *type* is a subtype of *is_a_type*.

`gtype-basic?` *type* [Primitive]
Returns `#t` if *type* is a basic type. Basic types have only one possible representation in Scheme. Unless the user means to deal in GValues, values of basic types should be manipulated as Scheme values.

`gtype-classed?` *type* [Primitive]
Returns `#t` if *type* is a classed type.

`gtype-instantiatable?` *type* [Primitive]
Returns `#t` if *type* is an instantiatable type.

`gtype-fundamental?` *type* [Primitive]
Returns `#t` if *type* is a fundamental type and `#f` if not. This is the same as `(eq? type (gtype->fundamental type))`, but slightly faster.

`gtype->fundamental` *type* [Primitive]
Returns the fundamental type of *type* (possibly *type* itself).

`gtype-parent` *type* [Primitive]
Returns the parent type of *type* (possibly *type* itself).

`gtype-children` *type* [Primitive]
Returns the `<gtype>`'s of *type*'s direct children, as a list.

`gtype-interfaces` *type* [Primitive]
Returns the set of `<gtype>`'s that *type* implements, as a list.

`gtype-name` *type* [Primitive]
Returns the name of *type*.

`gtype-from-name` *name* [Primitive]
Returns the type named *name*, or `#f` if none exists.

`gtype-from-instance` *instance* [Primitive]
Returns the type of *instance*, which should be a primitive `<%gtype-instance>`.

`%gtype-instance-primitive-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 not necessary. Used by the implementations of some instantiatable types that have `destroy` methods, notably `<gtk-object>`.

`gtype-instance-primitive->type` *instance* [Primitive]
Retrieve the `<gtype>` object associated with the primitive `<%gtype-instance>` value, *instance*.

`especify-metaclass!` *class metaclass* [Primitive]
A terrible hack that takes a class *class* and sets its metaclass, in-place, to *metaclass*. *metaclass* must be a subclass of *class*' existing metaclass.

This method is useful if you want to define a method that on a particular `<gtype-class>`, such as `make-instance`. However, it would be cleaner to devise a way of making these "class methods" without molesting GOOPS in this way.

`<gtype-class>` [Class]
The metaclass of all GType classes. Ensures that GType classes have `gtype` and `gtype-class` slots, which point to the primitive `<gtype>` and `<%gtype-class>` objects that wrap the C values.

`<gtype-instance-class>` [Class]
The metaclass of all instantiatable GType classes.

`<gtype-instance>` [Class]
The root class of all instantiatable GType classes. Adds a slot, `gtype-instance`, to instances. This slot will point to the primitive `<%gtype-instance>` object that wraps the C value.

`gtype->class` *type* [Function]
If there is already a GOOPS class associated with the GType *type*, return this class.

Otherwise, create a new GOOPS class and bind it to this type. The created class is an immortal, persistent object which is bound in some magic way to its GType.

`gtype-class->type` *class* [Function]
Returns the `<gtype>` associated with a `<gtype-class>`.

`%gtype-lookup-class` *type* [Primitive]
Returns the `<gtype-class>` registered for *type*, or `#f` if none has been registered.

`%gtype-bind-to-class` *class type* [Primitive]
A low-level procedure to bind the newly-created `<gtype-class>` *class* to *type*. Users should not need to call this function.

`gtype-instance:write` [Generic]
Generic function, defined so we can define `write` functions for instances of `<gtype-class>` in Scheme. A bit of a hack.

`gtype-instance:write` (*class* `<gtype-class>`) (*obj* `<%gtype-instance>`) (*file* `<top>`) [Method]

`gruntime-error` *format-string . args* [Function]
Signal a runtime error. The error will be thrown to the key `gruntime-error`.

`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"
```

`gtype:genum` [Variable]

`gtype:gflags` [Variable]

`gtype:ginterface` [Variable]

`gtype:gobject` [Variable]

`gtype:gparam` [Variable]

`gtype:void` [Variable]

2 (gnome gobject gvalue)

2.1 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>` are 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.

2.2 Usage

`gvalue?` *value* [Primitive]
Returns `#t` if *value* is a `<gvalue>`, `#f` otherwise.

`gvalue->type` *value* [Primitive]
Returns the `<gtype>` of the value held by *value*.

`<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. Use `gvalue->type` on an instance of this class to determine what type it holds.

`<gboxed-scm>` [Class]
A `<gboxed>` class for holding arbitrary Scheme objects.

`<gvalue-array>` [Class]
A `<gvalue>` class for arrays of `<gvalue>`.

`<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 from C either by passing the returned `<gtype>` object back to a C function or by using `g-type-from-name`.

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>` with name var{name} with the C runtime. See `genum-register-static` for details.

`genum-class->value-table` *class* [Function]
Return the vtable of possible values for *class*. The same as `genum-type-get-values`, but operates on classes.

`gflags-class->value-table` *class* [Function]
Return the vtable of possible values for *class*. The same as `gflags-type-get-values`, but operates on classes.

`genum-type-get-values` *type* [Primitive]
Return a vtable of the values supported by the enumerated `<gtype>` *type*. The return value will be in the format described in `genum-register-static`.

`gflags-type-get-values` *type* [Primitive]
Return a vtable of the values supported by the flag `<gtype>` *type*. The return value will be in the format described in `gflags-register-static`.

`scm->gvalue` *type scm* [Primitive]
Convert a Scheme value into a `<gvalue>` of type *type*. 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` *obj* [Function]
Convert the enumerated 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.

`gtype:gboolean` [Variable]

`gtype:gboxed` [Variable]

`gtype:gboxed-scm` [Variable]

`gtype:gchar` [Variable]

`gtype:gchararray` [Variable]

`gtype:gdouble` [Variable]

`gtype:gfloat` [Variable]

`gtype:gint` [Variable]

`gtype:gint64` [Variable]

`gtype:glong` [Variable]

`gtype:gpointer` [Variable]

`gtype:guchar` [Variable]

`gtype:guint` [Variable]

`gtype:guint64` [Variable]

`gtype:gulong` [Variable]

`gtype:gvalue-array` [Variable]

3 (gnome gobject gparameter)

3.1 Overview

Parameters are constraints for values, both in type and in range. This module wraps the parameters code of the GLib type system, defining C classes such that parameters may be manipulated and created from Scheme.

As a technical detail, the C structure `GParamSpec` is wrapped at two levels. One is a mapping of the C structure to a Guile structure. The other is a GOOPS representation. The low level is called `gparam-struct`, and the high level is called `<gparam>`. `gparam-struct` is a generic container of any type. `<gparam>` has subclasses for the various kinds of parameter types: `<gparam-int>`, `<gparam-object>`, etc.

3.2 Usage

`<gparam>` [Class]
The base class for GLib parameter objects.

`<gparam-char>` [Class]
Parameter for `<gchar>` values. 3 arguments: minimum, maximum, and default values.

`<gparam-uchar>` [Class]
Parameter for `<guchar>` values. 3 arguments: minimum, maximum, and default values.

`<gparam-boolean>` [Class]
Parameter for `<gboolean>` values. 1 argument: default value.

`<gparam-int>` [Class]
Parameter for `<gint>` values. 3 arguments: minimum, maximum, and default values.

`<gparam-uint>` [Class]
Parameter for `<guint>` values. 3 arguments: minimum, maximum, and default values.

`<gparam-long>` [Class]
Parameter for `<glong>` values. 3 arguments: minimum, maximum, and default values.

`<gparam-ulong>` [Class]
Parameter for `<gulong>` values. 3 arguments: minimum, maximum, and default values.

`<gparam-int64>` [Class]
Parameter for `<gint64>` values. 3 arguments: minimum, maximum, and default values.

`<gparam-uint64>` [Class]
Parameter for `<guint64>` values. 3 arguments: minimum, maximum, and default values.

`<gparam-float>` [Class]
Parameter for `<gfloat>` values. 3 arguments: minimum, maximum, and default values.

`<gparam-double>` [Class]
Parameter for `<gdouble>` values. 3 arguments: minimum, maximum, and default values.

`<gparam-pointer>` [Class]
Parameter for `<gpointer>` values. No arguments.

`<gparam-string>` [Class]
Parameter for `<gchararray>` values. 1 argument: the default value, which may be `#f`.

`<gparam-object>` [Class]
Parameter for `<gobject>` values. 1 argument: the `<gtype>` of the value.

`<gparam-boxed>` [Class]
Parameter for `<gboxed>` values. 1 argument: the `<gtype>` of the value.

`<gparam-enum>` [Class]
Parameter for `<genum>` values. 2 arguments: the `<gtype>` of the value, and the default value.

`<gparam-flags>` [Class]
Parameter for `<gflags>` values. 2 arguments: the `<gtype>` of the value, and the default value.

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

`gparam-struct:name` *param-struct* [Function]
Retrieve the name from a `gparam-struct`.

`gparam-struct:nick` *param-struct* [Function]
Retrieve the 'nickname' from a `gparam-struct`.

`gparam-struct:blurb` *param-struct* [Function]
Retrieve the 'blurb', a short descriptive string, from a `gparam-struct`.

`gparam-struct:flags` *param-struct* [Function]
Retrieve the flags from a `gparam-struct`.

`gparam-struct:param-type` *param-struct* [Function]
Retrieve the GParam type from a `gparam-struct`, for example `gtype:gparam-uint64`.

`gparam-struct:value-type` *param-struct* [Function]
Retrieve the value type from a `gparam-struct`, for example `gtype:guint64`.

`gparam-struct:owner-type` *param-struct* [Function]
Retrieve the 'owner type' from a `gparam-struct`. Appears to be stored into GLib param specs, but never used.

`gparam-struct:args` *param-struct* [Function]
Retrieve the arguments from a `gparam-struct`, as a list. The length and composition of the arguments depends on the parameter type.

`gparam->param-struct` *param* [Primitive]
Retrieve the primitive `gparam-struct` for the GOOPS parameter object, *param*.

`gparam->value-type` *param* [Primitive]
Retrieve the value type of the `<gparam>` object *param*.

`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]

`gtype:gparam-boolean` [Variable]

`gtype:gparam-boxed` [Variable]

gtype:gparam-char [Variable]
gtype:gparam-double [Variable]
gtype:gparam-enum [Variable]
gtype:gparam-flags [Variable]
gtype:gparam-float [Variable]
gtype:gparam-int [Variable]
gtype:gparam-int64 [Variable]
gtype:gparam-long [Variable]
gtype:gparam-object [Variable]
gtype:gparam-pointer [Variable]
gtype:gparam-string [Variable]
gtype:gparam-uchar [Variable]
gtype:gparam-uint [Variable]
gtype:gparam-uint64 [Variable]
gtype:gparam-ulong [Variable]

4 (gnome gobject gclosure)

4.1 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. This module exports a GOOPS class wrapping closures on the Scheme level, `<gclosure>`. `<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`. For example:

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

4.2 Usage

`<gclosure>` [Class]

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

`gclosure-invoke` *closure . args* [Function]

Invoke a closure. The arguments *args* will be converted to `<gvalue>` objects of the appropriate type, and the return value will be run through `gvalue->scm`. For all practical purposes, this function is like `apply`.

`gtype:gclosure` [Variable]

5 (gnome gobject gsignal)

5.1 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 instrospecting, emitting, connecting to, disconnecting from, blocking, and unblocking signals. Additionally it defines routines to define new signal types on instantiatable types.

5.2 Usage

`<gsignal>` [Variable]
The structure vtable for `<gsignal>` instances.

`gsignal:id` *signal* [Function]
Access the "id" of a `<gsignal>` structure, an integer.

`gsignal:name` *signal* [Function]
Access the name of a `<gsignal>` structure, a string.

`gsignal:interface-type` *signal* [Function]
Access the type to which a `<gsignal>` structure is associated, a `<gtype>`.

`gsignal:return-type` *signal* [Function]
Access the return type from a a `<gsignal>` structure, a `<gtype>`.

`gsignal:param-types` *signal* [Function]
Access the parameter types from a a `<gsignal>` structure, a list of `<gtype>`.

`gtype-get-signals` *type* [Primitive]
Returns a vector of signal names belonging to *type* and all parent types.

`gtype-class-get-signals` *class* [Function]
Returns a vector of signals belonging to *class* and all parent classes.

`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* [Function]
Emits signal *name* with arguments *args* on the `<gtype-instance>` *object*. *name* should be a symbol.

`gtype-instance-signal-connect-data` *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` *object name func* [Function]
Convenience function for calling `gtype-instance-signal-connect-data` with *after* = `#f`.

`gtype-instance-signal-connect-after` *object name func* [Function]
Convenience function for calling `gtype-instance-signal-connect-data` with *after* = `#t`.

`gsignal-handler-block` *obj id* [Function]
Block invocation of the signal handler identified by *id* from high-level GOOPS object *object*.

`gsignal-handler-unblock` *obj id* [Function]
Unblock invocation of the signal handler identified by *id* from high-level GOOPS object *object*.

`gsignal-handler-disconnect` *obj id* [Function]
Disconnect the signal handler identified by *id* from high-level GOOPS object *object*.

`gsignal-handler-connected?` *obj id* [Function]
Returns `#t` if the signal handler identified by *id* is connected on the high-level GOOPS object *object*, or `#f` otherwise.

`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 either a `<gtype>` or a `<gtype-class>` object. Similarly, *param-types* should be a list of either `<gtype>` or `<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.

`gtype-class-define-signal` [Special Form]
A macro invoked as:

```
(gtype-class-define-signal class name return-type
                           . param-types)
```

All arguments will be passed to `gtype-class-create-signal`.

This form is a macro because it will actually take the generic returned from `gtype-class-create-signal` and bind it to a name in the toplevel environment.

The name of the new generic function is the concatenation of the type name, a colon, and the signal name.

For example:

```
(gtype-class-define-signal <foo> 'roswell #f)
(define-method (foo:roswell (obj <foo>))
  *unspecified*)

(gtype-class-define-signal <foo> 'berlin <glong> <gint>)
(define-method (foo:berlin (obj <foo>) (x <number>))
  85)
```

6 (gnome gobject gobject)

6.1 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.

6.2 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>))
```

`<ginterface>` [Class]
The base class for GLib's interface types. Not derivable in Scheme.

`gtype-register-static` *name parent_type* [Primitive]
Derive a new type named *name* from *parent_type*. Returns the new `<gtype>`. 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.

`make-gobject-instance` [Generic]
A generic defined to initialize a newly created `<gobject>` instance. `make-gobject-instance` takes four arguments: the class of the object, its `<gtype>`, the object itself, and the options, which is a list of keyword arguments.

This operation is a generic function so that subclasses can override it, e.g. so that `<gtk-object>` can implement explicit destruction.

`make-gobject-instance` (*`class`* `<top>`) (*`type`* `<top>`) (*`object`* `<top>`) (*`options`* `<top>`) [Method]
The default implementation of `make-gobject-instance`.

`gobject-class-get-properties` *class* [Function]
Returns a vector of properties belonging to *class* and all parent classes.

`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* [Function]
Returns a vector of property names belonging to *class* and all parent classes.

`gobject-interface-get-properties` *class* [Function]
Returns a vector of properties belonging to *class* and all parent classes.

`gobject-interface-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-interface-get-property-names` *class* [Function]
Returns a vector of property names belonging to *class* and all parent classes.

`gobject-get-property` *object name* [Function]
Gets a the property named *name* (a symbol) from *object*.

`gobject-set-property` *object name init-value* [Function]
Sets the property named *name* (a symbol) on *object* to *init-value*.

7 (gnome gobject generics)

7.1 Overview

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

7.1.1 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))

;; 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.

7.1.2 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.

7.1.3 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.

7.2 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`.

8 (gnome gobject utils)

8.1 Overview

Common utility routines.

8.2 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-name->method-name` *type-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-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*.

9 (gnome gw support gobject)

9.1 Overview

Routines useful to *-spec.scm g-wrap files.

9.2 Usage

`<gobject-wrapset-base>` [Class]
The base class for G-Wrap wrapsets that use `<gobject>` types. Defines two slots, `type-aliases` and `type-rules`, to hold auxiliary type information.

`add-type-alias!` [Generic]

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

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

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

`add-type-rule!` [Generic]

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

`find-type-rule` [Generic]

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

`construct-argument-list` [Variable]
[unbound!]

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

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

`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-object-type>`) [Method]

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

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

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

`unwrap-null-checked` *value status-var code* [Function]

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-object!` [Generic]

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

`wrap-boxed!` [Generic]

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

`wrap-pointer!` [Generic]

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

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

`wrap-interface!` [Generic]

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

`wrap-flags!` [Generic]

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

`wrap-gobject-class!` [Generic]

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

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

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

10 (gnome gw support defs)

10.1 Overview

Support for reading in Gtk .defs files as g-wrap instructions

10.2 Usage

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

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

11 (gnome gw support gtk-doc)

11.1 Overview

Parsing a subset of the docbook emitted by gtk-doc into `stexi`.

11.2 Usage

`docbook->sdocbook` *docbook-fragment* [Function]
Parse a docbook file *docbook-fragment* into SXML. Simply calls SSAX's `xml->sxml`, but having made sure that '` `' elements are interpreted correctly. Does not deal with XInclude.

`gtk-doc-sdocbook-title` *sdocbook* [Function]
Extract the title from a fragment of docbook, as produced by gtk-doc. May return `#f` if the title is not found.

`gtk-doc-sdocbook-subtitle` *sdocbook* [Function]
Extract the subtitle from a fragment of docbook, as produced by gtk-doc. May return `#f` if the subtitle is not found.

`gtk-doc-sdocbook->description-fragment` *sdocbook* [Function]
Extract the "description" of a module from a fragment of docbook, as produced by gtk-doc, translated into texinfo.

`gtk-doc-sdocbook->def-list` *sdocbook process-def* [Function]
Extract documentation for all functions defined in the docbook nodeset *sdocbook*.

When a function is found and translated into texinfo, *process-def* will be called with two arguments, the name of the procedure as a symbol, and the documentation as a `deffn`. *process-def* may return `#f` to indicate that the function should not be included in the documentation; otherwise, the return value of *process-def* will be used as the documentation.

This mechanism allows the caller of `gtk-doc-sdocbook->def-list` to perform further processing on the documentation, including the possiblity of replacing it completely with documenation from another source, for example a file of hand-written documentation overrides.

12 (gnome gw support modules)

12.1 Overview

This module implements some procedures useful to modules that use g-wrapped libraries.

12.2 Usage

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

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

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

Concept Index

(Index is nonexistent)

Function Index

(Index is nonexistent)