

# Speeding Up PyGObject Introspection

**Version: 0.1 draft**

**Author: John (J5) Palmieri <johnp@redhat.com>**

## **Abstract**

PyGObject Introspection (PyGI) is a system that reads metadata from GObject based libraries and is able to call into them through Python without compiled glue code in between. This means that any library that is marked up with introspection data can instantly be used within python. This comes with some drawbacks. Since the introspection metadata needs to be rich in order to describe a variety of interfaces, including legacy code, there is a fair amount of processing overhead that needs to be done with each call. While we have tightened the patterns that are considered to be wrappable via introspection, there are still numerous cases that need to be handled.

This doesn't mean we have to be slow, just that it is easier to write an invoker that is slow. Right now our invoker is in need of some cleanup. It is currently an order of magnitude slower than hand written bindings. We end up looping over the argument list a number of times and have no real caching to speak of. I propose with a proper evaluation of the invoke chain we can speed up current invocations by significant amounts. A current stress test calling methods with basic parameter lists produces this output on my machine which can be used as a baseline to compare performance (note these torture tests don't really hit the slow bits but are the common case):

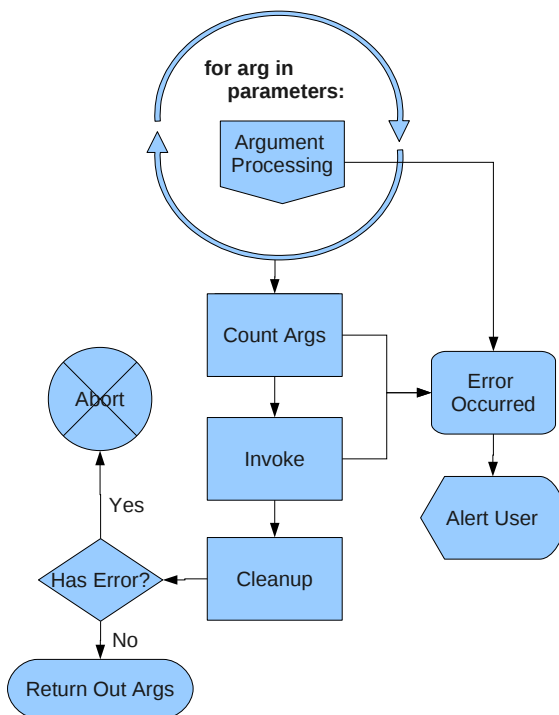
```
test_torture_profile (test_everything.TestTortureProfile) ...
  torture test 1 (10000 iterations): 0.240000 secs
  torture test 2 (10000 iterations): 0.380000 secs
  torture test 3 (10000 iterations): 0.260000 secs
  torture test 4 (10000 iterations): 0.400000 secs
====
Total: 1.280000 sec
```

## **Goals**

- Map out the entire invoke state machine
- Reduce argument validation and marshaling to one pass over the parameter list (we might need a metadata pass due to some information not being complete in the typelib)
- Identify cachable attributes (those that do not change with each call)
- Propose a way of caching the validation and marshaling routines so on subsequent calls we simply loop over the parameter list calling the cached validation and marshaling functions
- Make sure those routines have as little decision branches (if, switch, etc.) as possible
- Make sure those routines do not need to call into GObject Introspection at all (e.g. they have enough information to do the job without asking outside libraries)
- Simplify the code so it is easier to understand and modify

## Fixing Invoke

### Invoke Overview



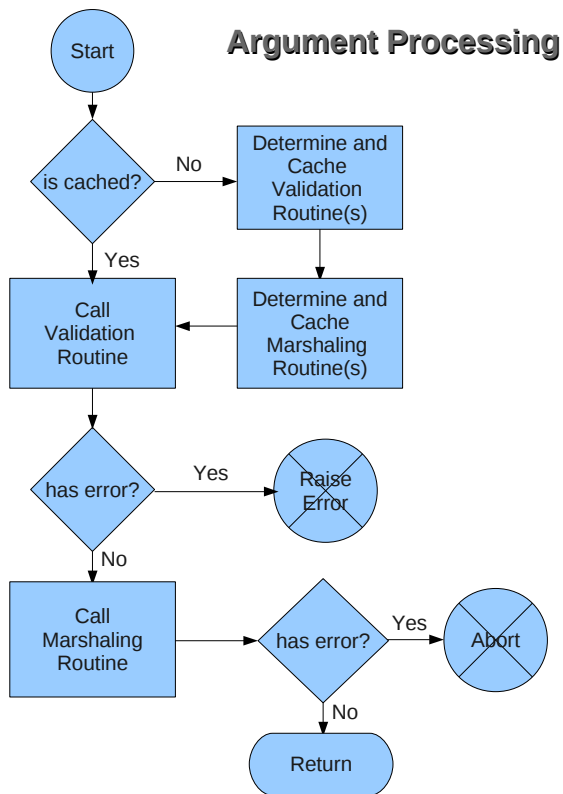
Invoke is the entry point that gets called whenever we need to translate a Python call down to a GI native call. It is here where we check the parameters and marshal them into formats the underlying GObject libraries will understand. We then take the results and marshal them into Python objects.

Right now Invoke splits the processing into four steps – validation, marshalling, the actual invocation and cleanup. Each of these steps may loop over the parameter list and some of them loop over it several times to handle special cases. An issue arises because these steps aren't always strict in their purpose. For instance code has been added which does validation during the marshaling phase.

The figure above diagrams an overview of how I think invoke should look. There is one loop over the arguments where we validate and marshal them. GI has all the positional information needed to process axillary parameter edge cases

(special parameters that provide metadata for other parameters or are handled directly by PyGI, such as GErrors) so we don't need loop over the argument list more than once during the argument processing stage. *Note that right now this is idealistic. The typelib does not give us all of the metadata we need for special cases such as length/array combos where the length argument comes before the array. We can add a loop to gather the needed metadata leaving the rest of the design intact.*

In the preceding pages I will jump down the rabbit hole and further detail out each stage of the state tree that is needed to implement an efficient invoke procedure. This can be used by anyone wishing to implement GI for their language of interest. I will however be concentrating on Python and the PyGI implementation.



## Argument Processing Overview

Now that we have seen the general high level overview of invoke, we can now start to dig down into the details. Here you will see a diagram of how I am proposing we break up the processing of each argument in a parameter list. Later on we will tackle each type of parameter that we can handle, including all edge cases currently known.

Argument processing is the heart of the Invoke process. This is where most of the slowdowns occur and calling into GI, with all the branching that occurs, has the potential to produce user noticeable slowdowns.

The diagram on this page specifically deals with this by caching the validation and marshaling routines the first time the interface is run. Subsequent calls simply run those functions on the argument data.

The goal here is to keep all the branching and GI calls exclusive to determining the cached

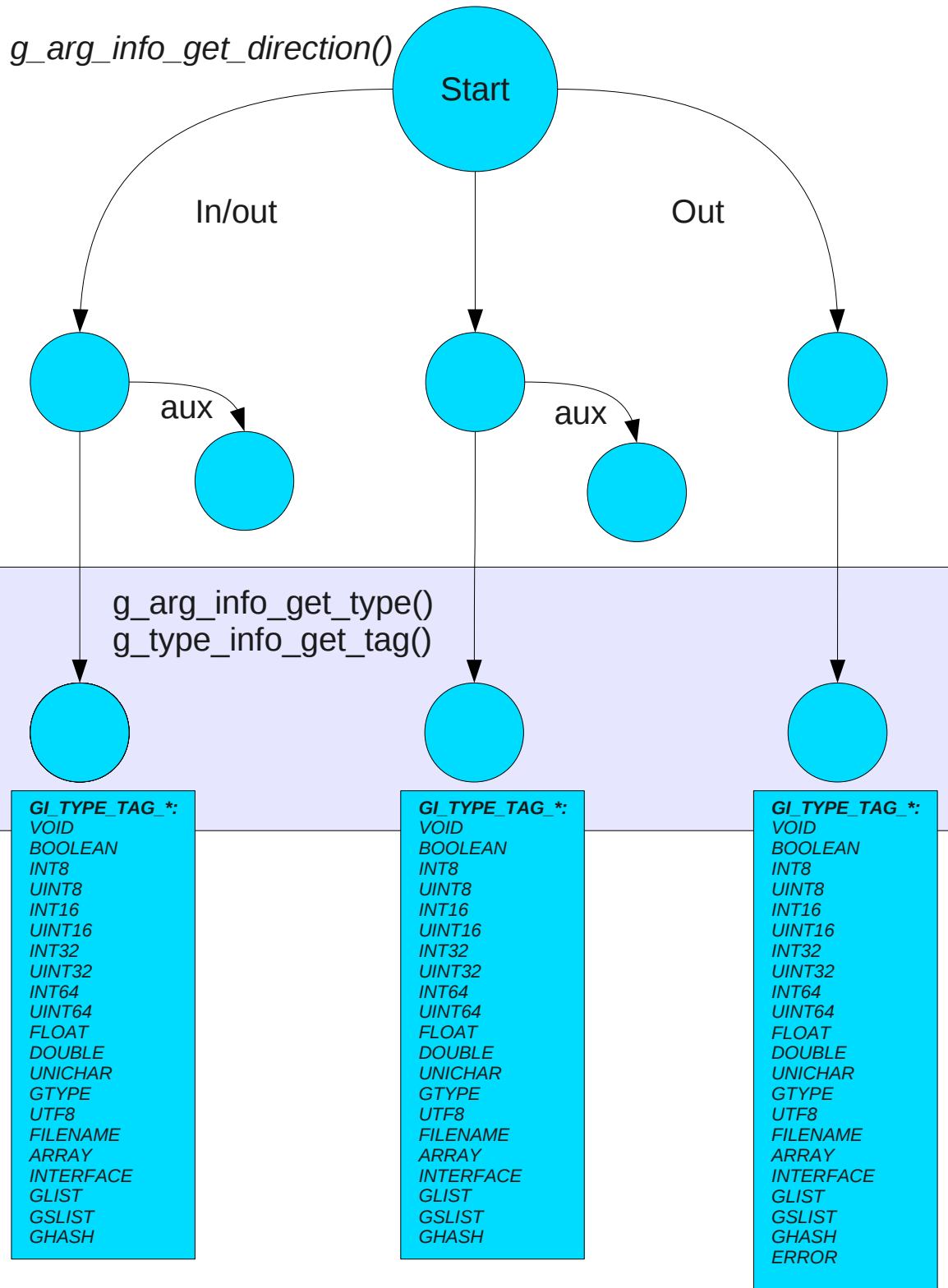
functions. While validation and marshaling aren't without their own overheads, this significantly reduces them. Another goal here is to break up the validation and marshaling into small functions for each parameter type we encounter as opposed to the huge switch statements we have now. This will make it easier to fix bugs as well as add new types as we encounter them.

A validation function should have the following signature:

```

/* FIXME: this is not the final signature */
gboolean (*PyGIVValidationFunc) (PyObject *param, GIInvokeState *state);
  
```

The function should modify the state accordingly and marshal the parameter returning TRUE on success and FALSE if a python error is set.



## Handling Types

In the diagram that precedes this section you will see a simplified state diagram for processing each argument type. We will now go into depth on each of the types listed.

## Aux Parameters

Auxiliary *in* parameters are those which are present in the C interface but which we do not have to present to the user because we can fill in the data in some other way. We skip processing these values when we encounter them. Their associated parameters are in charge of validating and marshaling them.

- **Array length** – there are a number of ways an array can specify its length. Either it is null terminated, has a fixed size or uses a parameter to send the length. Since we always know the size of our arrays in python we don't let the user set the array length parameter and instead set it ourselves
  - *Relevant API* – `g_arg_info_get_array_length` – called on the array's arg info
- **GDestroyNotify** – This pointer to a function is sent in to manage the lifecycle of a callback. When the callback needs to be destroyed the library will call the GDestroyNotify function so that the app can clean up its callback. We pass in a generic GDestroyNotify so the user does not need to (an in fact this is a Cism that would not be useful within Python itself)
  - *Relevant API* – `g_arg_info_get_destroy` – called on the callback func's arg info
- **User Data** – This parameter is optional if it is the last *in* parameter otherwise it is mandatory. In a sense it is optionally aux.
  - *Relevant API* – `g_arg_info_get_closure` – called on the callback func's arg info

## Special Arguments

There are a couple of arguments we special case and fall somewhat outside the realm of the normal argument handling and aux argument handling, so should be mentioned here.

- **Self** – This is just like any other argument but is not represented in the argument list sent back from GI. On any call marked as a method (as opposed to a function), the first argument will always be the instance object we are calling the method on. We check this the same way we do other *in* arguments with one exception – it only needs to be validated on first call (we may validate each call for debugging purposes but this allows us to do a simple optimization and be sure, beyond data corruption, that the object is already validated). Since it does not show up in the argument list implementors must make sure to not count the self python object as part of the normal argument list when matching type infos to the python value passed in the arg list.
  - *Relevant API*:
    - `g_function_info_get_flags(function_info) & GI_FUNCTION_IS_METHOD` – called on the function info, lets you know if this function is a method and has an instance object
    - `container_info = g_base_info_get_container(function_info)` - called on the function info, gets the info (similar to arg info) for the instance type

- `container_info_type = g_base_info_get_type (container_info)` – called on the container info, gets the `GI_TYPE_TAG` for the instance
- *Supported Types* – these types can have methods associated with them:
  - `GI_TYPE_TAG_UNION`
  - `GI_TYPE_TAG_STRUCT`
  - `GI_TYPE_TAG_OBJECT`
  - `GI_TYPE_TAG_INTERFACE`
- **Error** – GObject API signifies errors via a passed in GError object as the last parameter. We could treat this as a normal *out caller allocates* value but most implementations want to be able to handle errors in a much more integrated way. In python we clean up the state and throw the error as a normal python error
- **Defaults** – While not a specific argument and not fully implemented in GI, we none the less want to be able to support defaults in the future. Defaults allow *in* parameters to be left out and simply take a default value.
- **Return** – like an out value except this is returned from the function instead of as a parameter of the function. If there are no out values the return value is the only thing returned. If there are out values the return value is returned as the first value of a tuple. The return value can be of type `VOID (GI_TYPE_TAG_VOID)` in which case there is no return value. If there is no return value and no out values `None` is returned.
  - *Relevant API:*
    - `g_callable_info_get_return_type(callable_info)` – called on the function info cast to a callable info, returns the type info
    - `g_type_info_get_tag(type_info)` – called on the type info returns a `GI_TYPE_TAG`

## In Arguments

*In* arguments are parameters which the app developer passes to the function or method. When counting arguments, the number of python arguments (`py_arg_n`) must match the number of *in* arguments (`in_arg_n`) where `in_arg_n = number of args – out args – aux args – error arg – default args`.

In arguments are processed like this in GI:

```
int n_args = g_callable_info_get_n_args ( (GICallableInfo *) function_info);
for (i = 0; i < n_args; i++) {
    GIArgInfo *arg_info =
        g_callable_info_get_arg ( (GICallableInfo *) function_info, i);
    GIDirection direction = g_arg_info_get_direction (arg_info);
    GITypeInfo *type_info = g_base_info_get_type ( (GIBaseInfo *) arg_info);
    GITypeTag type_tag = g_type_info_get_tag (type_info);

    if (arg_cache->is_aux)
        continue;

    switch(direction) {
        case GI_DIRECTION_IN:
            function_cache->in_args_n++;
            switch (type_tag) {
                case GI_TYPE_TAG_....:
                    /* we could also save a call by making the validator and
                     * marshaler inline functions and have a wrapper become
                     * the cached pointer */

```

```

        arg_cache->in_validator = <type validation function pointer>
        arg_cache->in_marshaler = <type marshaling function pointer>
        break;
    }
    break;
    .
    .
}

```

### ***GI\_TYPE\_TAG\_VOID***

Void is an undefined type which we just pass on as a pointer.

#### **Validation:**

none

#### **Marshaling:**

```

g_warn_if_fail (transfer == GI_TRANSFER_NOTHING);
arg.v_pointer = py_object;

```

### ***GI\_TYPE\_TAG\_BOOLEAN***

True/False value

#### **Validation:**

No check; every Python object has a truth value.

#### **Marshaling:**

```

arg.v_boolean = PyObject_IsTrue (py_object);

```

### ***GI\_TYPE\_TAG\_INT8***

A 8 bit signed interger

#### **Validation:**

```

PyNumber_Check (py_object)
lower_bounds = -128
upper_bounds = 128

```

#### **Marshaling:**

### ***GI\_TYPE\_TAG\_UINT8***

A 8 bit unsigned integer or single ascii character

#### **Validation:**

```

PyNumber_Check (py_object)
lower_bounds = 0
upper_bounds = 255

```

#### **Marshaling:**

### ***GI\_TYPE\_TAG\_INT16***

A 16 bit signed integer

**Validation:**

PyNumber\_Check (py\_object)  
lower\_bounds = -32768  
upper\_bounds = 32767

**Marshaling:**

***GI\_TYPE\_TAG\_UINT16***

A 16 bit unsigned integer

**Validation:**

PyNumber\_Check (py\_object)  
lower\_bounds = 0  
upper\_bounds = 65535

**Marshaling:**

***GI\_TYPE\_TAG\_INT32***

A 32 bit signed integer

**Validation:**

PyNumber\_Check (py\_object)  
lower\_bounds = G\_MININT32  
upper\_bounds = G\_MAXINT32

**Marshaling:**

***GI\_TYPE\_TAG\_UINT32***

A 32 bit unsigned integer

**Validation:**

PyNumber\_Check (py\_object)  
lower\_bounds = 0  
upper\_bounds = G\_MAXUINT32

**Marshaling:**

***GI\_TYPE\_TAG\_INT64***

A 64 bit signed integer

**Validation:**

PyNumber\_Check (py\_object)  
lower\_bounds = G\_MININT64  
upper\_bounds = G\_MAXINT64

**Marshaling:**

**GI\_TYPE\_TAG\_UINT64**

A 64 bit unsigned integer

**Validation:**

PyNumber\_Check (py\_object)  
lower\_bounds = 0  
upper\_bounds = G\_MAXUINT64

**Marshaling:****GI\_TYPE\_TAG\_FLOAT**

A single precision floating point type

**Validation:**

PyNumber\_Check (py\_object)  
lower\_bounds = -G\_MAXFLOAT  
upper\_bounds = G\_MAXFLOAT

**Marshaling:****GI\_TYPE\_TAG\_DOUBLE**

A double precision floating point type

**Validation:**

PyNumber\_Check (py\_object)  
lower\_bounds = -G\_MAXDOUBLE  
upper\_bounds = G\_MAXDOUBLE

**Marshaling:****GI\_TYPE\_TAG\_UNICHAR**

A single unicode character

**Validation:**

PyUnicode\_Check (py\_object)  
PyString\_Check (object) /\* Python 2 \*/  
size == 1

**Marshaling:****GI\_TYPE\_TAG\_GTYPE**

A GObject (type registered with the glib type system)

**Validation:**

pyg\_type\_from\_object (py\_object) != 0

**Marshaling:****GI\_TYPE\_TAG\_UTF8**

A unicode string

**Validation:**

```
PYGLIB_PyBaseString_Check (py_object) /* PyString or PyUnicode */
```

**Marshaling:**

***GI\_TYPE\_TAG\_FILENAME***

A unicode string representing a filename

**Validation:**

```
PYGLIB_PyBaseString_Check (py_object) /* PyString or PyUnicode */
```

**Marshaling:**

***GI\_TYPE\_TAG\_ARRAY***

A sequence of values

**Validation:**

```
PySequence_Check (py_object)
length > -1
if arg_cache->array_cache->fixed_size:
    length <= fixed_size
```

```
first_item = PySequence_GetItem (py_object, 0);
arg_cache->sequence_cache->item_validate_func(first_item)
```

if DEBUG:

```
/* check every item */
for (i = 1; i < length; i++):
    item = PySequence_GetItem (py_object, i);
    arg_cache->sequence_cache->item_validate_func(item);
```

**Marshaling:**

***GI\_TYPE\_TAG\_INTERFACE***

An interface to a complex type. On first pass we cache separate validation and marshaling functions for each interface type:

```
info_type = g_base_info_get_type (info);
arg_cache->interface_cache->info = info;
arg_cache->interface_cache->g_type = g_registered_type_info_get_g_type (info);
arg_cache->interface_cache->py_type = _pygi_type_import_by_gi_info (info);
```

**Validation:**

```
GI_INFO_TYPE_CALLBACK:
    PyCallable_Check (py_object)
```

```
GI_INFO_TYPE_ENUM:
```

```

if PyNumber_Check(py_object):
    /* use GI here since caching all enum values is inefficient */
    number == 0;
    if DEBUG:
        n_enums = g_enum_info_get_n_values (info);
        loop:
            value_info = g_enum_info_get_value (arg_cache->interface_cache->info, i);
            enum_value = g_value_info_get_value (value_info);
            enum_value == number

else:
    PyObject_IsInstance(py_object, arg_cache->interface_cache->py_type)

```

**GI\_INFO\_TYPE\_FLAGS:**

```

if PyNumber_Check(py_object):
    number == 0;

```

else:

```

PyObject_IsInstance(py_object, arg_cache->interface_cache->py_type)

```

**GI\_INFO\_TYPE\_STRUCT:**

```

/* cache validation routine as struct can be evaluated to other types */
if g_type_is_a(arg_cache->interface_cache->g_type, G_TYPE_CLOSURE):
    GI_INFO_TYPE_CALLBACK_validate()
elif g_type_is_a(arg_cache->interface_cache->g_type, G_TYPE_VALUE):
    /* not enough context to validate */

```

```

PyObject_IsInstance(py_object, arg_cache->interface_cache->py_type)

```

**GI\_INFO\_TYPE\_BOXED:**

**GI\_INFO\_TYPE\_INTERFACE:**

**GI\_INFO\_TYPE\_OBJECT:**

**GI\_INFO\_TYPE\_UNION:**

```

PyObject_IsInstance(py_object, arg_cache->interface_cache->py_type)

```

## **Marshaling:**

### ***GI\_TYPE\_TAG\_GLIST***

A doubly linked list represented by any python sequence

#### **Validation:**

```

GI_TYPE_TAG_ARRAY_validate()

```

## **Marshaling:**

### ***GI\_TYPE\_TAG\_GSLIST***

A singly linked list represented by any python sequence

#### **Validation:**

```

GI_TYPE_TAG_ARRAY_validate()

```

## Marshaling:

### ***GI\_TYPE\_TAG\_GHASH***

A dictionary represented by any PyObject supporting the Mapping interface

#### **Validation:**

```
keys = PyMapping_Keys (py_object);
keys != NULL;
length = PyMapping_Length (py_object);
length >= 0;
values = PyMapping_Values (object);
values != NULL;

first_key = PySequence_GetItem (keys, 0);
first_value = PySequence_GetItem (values, 0);
arg_cache->hash_cache->key_validate_func(first_key);
arg_cache->hash_cache->value_validate_func(first_value);

if DEBUG:
    /* check every item */
    for (i = 1; i < length; i++):
        key_item = PySequence_GetItem (keys, i);
        value_item = PySequence_GetItem (values, i);
        arg_cache->hash_cache->key_validate_func(key_item);
        arg_cache->hash_cache->value_validate_func(value_item);
```

## Marshaling:

### ***GI\_TYPE\_TAG\_ERROR***

should not reach

While errors could theoretically be supported there really is no reason to be directly manipulating GErrors in a scripting language so we won't support it unless someone can give a good argument on why we should.

## Out Arguments

*Out* arguments are parameters which are returned to the app developer. There are two types of out arguments:

- *callee allocates* – this is where a double pointer is passed and the function being called sends back a pointer along with a scope of the value passed back. Depending on the scope the value may need to be cleaned up.
- *caller allocates* – this is where the app developer would normally allocate an empty structure on the stack and pass it into the function which would then fill in the values. Since we are generating the code dynamically and we don't want the programmer to worry about the difference between callee and caller allocates we must allocate the structure on the heap and make sure it is cleaned up properly when the python wrapper is destroyed.

In arguments are processed like this in GI:

```
int n_args = g_callable_info_get_n_args ( (GICallableInfo *) function_info);
for (i = 0; i < n_args; i++) {
    GIArgInfo *arg_info =
        g_callable_info_get_arg ( (GICallableInfo *) function_info, i);
    GIDirection direction = g_arg_info_get_direction (arg_info);
    GITypeInfo *type_info = g_base_info_get_type ( (GIBaseInfo *) arg_info);
    GITypeTag type_tag = g_type_info_get_tag (type_info);

    switch(direction) {
        case GI_DIRECTION_OUT:
            function_cache->in_args_n++;
            switch (type_tag) {
                case GI_TYPE_TAG_....:
                    /* we could also save a call by making the validator and
                     * marshaler inline functions and have a wrapper become
                     * the cached pointer */
                    arg_cache->out_validator = <type validation function pointer>
                    arg_cache->out_marshaler = <type marshaling function pointer>
                    arg_cache->out_cleanup = <type cleanup function pointer>
                    break;
            }
            break;
        .
        .
        .
    }
}
```

```
GI_TYPE_TAG_VOID
GI_TYPE_TAG_BOOLEAN
GI_TYPE_TAG_INT8
GI_TYPE_TAG_UINT8
GI_TYPE_TAG_INT16
GI_TYPE_TAG_UINT16
GI_TYPE_TAG_INT32
GI_TYPE_TAG_UINT32
GI_TYPE_TAG_INT64
GI_TYPE_TAG_UINT64
GI_TYPE_TAG_FLOAT
GI_TYPE_TAG_DOUBLE
GI_TYPE_TAG_UNICHAR
GI_TYPE_TAG_GTYPE
GI_TYPE_TAG_UTF8
GI_TYPE_TAG_FILENAME
GI_TYPE_TAG_ARRAY
GI_TYPE_TAG_INTERFACE
GI_TYPE_TAG_GLIST
GI_TYPE_TAG_GSLIST
GI_TYPE_TAG_GHASH
```