# JuliaInterface

## Interface to Julia

0.5.2

14 February 2021

**Thomas Breuer**

**Sebastian Gutsche**

**Max Horn**

**Thomas Breuer**
Email: sam@math.rwth-aachen.de
Homepage: http://www.math.rwth-aachen.de/~Thomas.Breuer
Address: Thomas Breuer
Lehrstuhl für Algebra und Zahlentheorie
RWTH Aachen
Pontdriesch 14/16
52062 Aachen
Germany

**Sebastian Gutsche**
Email: gutsche@mathematik.uni-siegen.de
Homepage: https://sebasguts.github.io
Address: Department Mathematik
Universität Siegen
Walter-Flex-Straße 3
57072 Siegen
Germany

**Max Horn**
Email: horn@mathematik.uni-kl.de
Homepage: https://www.quendi.de/math
Address: Fachbereich Mathematik
TU Kaiserslautern
Gottlieb-Daimler-Straße 48
67663 Kaiserslautern
Germany

## Abstract

The GAP package JuliaInterface is part of a bidirectional interface between GAP and Julia.



## Acknowledgements

The development of this GAP package has been supported by the SFB-TRR 195 “Symbolic Tools in Mathematics and their Applications” (from 2017 until 2021).

# Contents

# Chapter 1

# Introduction to JuliaInterface

The GAP package JuliaInterface is part of a bidirectional interface between GAP and Julia.

## 1.1 Aims of the JuliaInterface package

The low level interface between GAP and Julia allows one to access GAP objects and to call GAP functions in a Julia session, to access Julia objects and to call Julia functions in a GAP session, and to convert low level data such as integers, booleans, strings, arrays/lists, dictionaries/records between the two systems.

In particular, this interface is *not* intended to provide a very "Julia-ish" interface to GAP objects and functions, nor a "GAP-ish" interface to Julia objects and functions.

Also, the interface does not provide conversions to GAP for Julia objects whose types are defined in Julia packages (that is, not in the "core Julia"). For example, the Julia package Nemo.jl defines an integer type `fmpz`. The conversion of integers of type `fmpz` between Julia and GAP is handled in the context of the Oscar system, which uses both GAP and Nemo.jl, but JuliaInterface does not deal with it.

The interface consists of

- the integration of Julia's garbage collector into GAP (which belongs to the GAP core system),
- `C` code for converting and wrapping low level objects (which belongs to JuliaInterface),
- Julia code for converting low level objects (which belongs to the Julia package `GAP.jl`, see https://github.com/oscar-system/GAP.jl),
- and GAP code (again in JuliaInterface) which is described in this manual.

The JuliaInterface manual takes the viewpoint of a GAP session from where one wants to use Julia functionality. The opposite direction, using GAP functionality in a Julia session, is described in the documentation of the Julia package `GAP.jl`.

## 1.2 Installation of the JuliaInterface package

The package can be used only when the underlying GAP has been compiled with the Julia garbage collector, and the recommended way to install such a GAP is to install Julia first (see

https://julialang.org/downloads/) and then to ask Julia's package manager to download and install GAP, by entering

Julia
```
using Pkg; Pkg.add( "GAP" )
```

at the Julia prompt.

One way to start a GAP session from the Julia session is to enter

Julia
```
using GAP; GAP.prompt()
```

at the Julia prompt, afterwards the package JuliaInterface is already installed and loaded.

Alternatively, one can start GAP in the traditional way, by executing a shell script. Such a script is generated automatically during the installation of GAP via Julia, its location is returned in a Julia session by

Julia
```
using GAP; GAP.gap_exe()
```

Note that the JuliaInterface code belongs to the Julia package `GAP.jl`, hence it can be found there.

## 1.3 User preferences in the JuliaInterface package

### 1.3.1 User preference `IncludeJuliaStartupFile`

When one starts an interactive Julia session, the Julia startup file `~/.julia/config/startup.jl` gets included automatically by default, see the section "The Julia REPL" in the Julia documentation. Hence the effects of this inclusion can be used in a GAP session which one gets via the following input.

Julia
```
using GAP; GAP.prompt()
```

However, this Julia startup file is *not* included into Julia by default when GAP gets started via the `gap.sh` script that is created during the installation of GAP (controlled by Julia).

The user preference `IncludeJuliaStartupFile` can be used to force that the startup file gets included also in the latter situation, as follows.

If the value is `true` then the file `~/.julia/config/startup.jl` gets included into Julia after startup. If the value is a nonempty string that is the name of a directory then the `startup.jl` file in this directory gets included into Julia. Otherwise (this is the default) no `startup.jl` file will be included automatically.

# Chapter 2

# Using Julia from GAP

## 2.1 Filters for JuliaInterface

### 2.1.1 IsJuliaObject (for IsObject)

▷ `IsJuliaObject(`*obj*`)` (filter)

**Returns:** `true` or `false`

The result is `true` if and only if *obj* is a pointer to a Julia object.

The results of `JuliaModule` (2.3.4) are always in `IsJuliaObject`. The results of `JuliaEvalString` (2.2.1) are in `IsArgumentForJuliaFunction` (2.1.5) but not necessarily in `IsJuliaObject`.

Example

```
gap> julia_fun:= JuliaEvalString( "sqrt" );
<Julia: sqrt>
gap> IsJuliaObject( julia_fun );
true
gap> julia_val:= julia_fun( 2 );
<Julia: 1.4142135623730951>
gap> IsJuliaObject( julia_val );
true
gap> julia_x:= JuliaEvalString( "x = 4" );
4
gap> IsJuliaObject( julia_x );
false
gap> IsJuliaObject( JuliaModule( "Main" ) );
true
```

### 2.1.2 IsJuliaWrapper (for IsObject)

▷ `IsJuliaWrapper(`*obj*`)` (filter)

**Returns:** `true` or `false`

If the component or positional object *obj* is in this filter then calling a Julia function with *obj* as an argument will not pass *obj* as an `MPtr`, but instead its `JuliaPointer` (2.1.3) value is passed, which must be a Julia object. This admits implementing high-level wrapper objects for Julia objects that behave just like the Julia objects when used as arguments in calls to Julia functions.

Objects in `IsJuliaWrapper` should *not* be in the filter `IsJuliaObject` (2.1.1).

Examples of objects in IsJuliaWrapper are the return values of JuliaModule (2.3.4).

### 2.1.3 JuliaPointer (for IsJuliaWrapper)

▷ JuliaPointer(*obj*) (attribute)

is an attribute for GAP objects in the filter IsJuliaWrapper (2.1.2). The value must be a Julia object.

Example

```
gap> Julia;
<Julia module Main>
gap> IsJuliaObject( Julia );
false
gap> IsJuliaWrapper( Julia );
true
gap> ptr:= JuliaPointer( Julia );
<Julia: Main>
gap> IsJuliaObject( ptr );
true
```

### 2.1.4 IsJuliaModule (for IsJuliaWrapper)

▷ IsJuliaModule(*obj*) (filter)

**Returns:** true or false

This filter is set in those GAP objects that represent Julia modules. A submodule of a module can be accessed like a record component, provided that this submodule has already been imported, see ImportJuliaModuleIntoGAP (2.3.3). Julia variables from a module can be accessed like record components.

Example

```
gap> IsJuliaModule( Julia );
true
gap> Julia.GAP;
<Julia module GAP>
gap> IsJuliaModule( Julia.GAP );
true
gap> Julia.GAP.julia_to_gap;
<Julia: julia_to_gap>
gap> JuliaFunction( "julia_to_gap", "GAP" );  # the same function
<Julia: julia_to_gap>
```

### 2.1.5 IsArgumentForJuliaFunction

▷ IsArgumentForJuliaFunction(*obj*) (function)

This function returns true for all those GAP objects that can be used as arguments of Julia functions. These are the objects in IsJuliaObject (2.1.1), IsJuliaWrapper (2.1.2), IsBool (**Reference: IsBool**), IsInt and IsSmallIntRep (see (**Reference: Integers**)), and IsFFE and IsInternalRep (see IsFFE (**Reference: IsFFE**)). For all other GAP objects, the function returns false.

Example

```
gap> m:= JuliaEvalString( "sqrt(2)" );;
gap> ForAll( [ m, Julia, true, 1, Z(2) ], IsArgumentForJuliaFunction );
true
gap> ForAny( [ 2^62, [], (1,2) ], IsArgumentForJuliaFunction );
false
```

## 2.2 Creating Julia objects

### 2.2.1 JuliaEvalString

▷ `JuliaEvalString(`*`string`*`)` (function)

evaluates the string *`string`* in the current Julia session, in the `Main` module, and returns Julia's return value.

Example

```
gap> JuliaEvalString( "x = 2^2" );  # assignment to a variable in Julia
4
gap> JuliaEvalString( "x" );        # access to this variable
4
```

### 2.2.2 JuliaIncludeFile

▷ `JuliaIncludeFile(`*`filename[, module_name]`*`)` (function)

**Returns:** nothing.

calls Julia's `Base.include` with the strings *`filename`* (an absolute filename, as returned by `Filename` (**Reference: Filename**)) and *`module_name`* (the name of a Julia module, the default is `"Main"`). This means that the Julia code in the file with name *`filename`* gets executed in the current Julia session, in the context of the Julia module *`module_name`*. If the file defines a new Julia module then the next step will be to import this module, see `ImportJuliaModuleIntoGAP` (2.3.3).

### 2.2.3 JuliaImportPackage

▷ `JuliaImportPackage(`*`pkgname`*`)` (function)

**Returns:** `true` or `false`.

This function triggers the execution of an `import` statement for the Julia package with name *`pkgname`*. It returns `true` if the call was successful, and `false` otherwise.

Note that we just want to load the package into Julia, we do *not* want to import variable names from the package into Julia's `Main` module, because the Julia variables must be referenced relative to their modules if we want to be sure to access the correct values.

Why is this function needed?

Apparently `libjulia` throws an error when trying to compile the package, which happens when some files from the package have been modified since compilation.

Thus GAP has to check whether the Julia package has been loaded successfully, and can then safely load and execute code that relies on this Julia package. In particular, we cannot just put the necessary `import` statements into the relevant `.jl` files, and then load these files with `JuliaIncludeFile` (2.2.2).

## 2.3 Access to Julia objects

### 2.3.1 JuliaFunction

▷ JuliaFunction(*function_name[, module_name]*) (function)

**Returns:** a function

Returns a GAP function that wraps the Julia function with identifier *function_name* from the module *module_name*. Both arguments must be strings. If *module_name* is not given, the function is taken from Julia's Main module. The returned function can be called on arguments in IsArgumentForJuliaFunction (2.1.5).

Example

```
gap> fun:= JuliaFunction( "sqrt" );
<Julia: sqrt>
gap> Print( fun );
function ( arg... )
    <<kernel code>> from Julia:sqrt
end
gap> IsFunction( fun );
true
gap> IsJuliaObject( fun );
false
```

Alternatively, one can access Julia functions also via the global object Julia (2.3.2), as follows.

Example

```
gap> Julia.sqrt;
<Julia: sqrt>
```

Note that each call to JuliaFunction and each component access to Julia (2.3.2) create a *new* GAP object.

Example

```
gap> IsIdenticalObj( JuliaFunction( "sqrt" ), JuliaFunction( "sqrt" ) );
false
gap> IsIdenticalObj( Julia.sqrt, Julia.sqrt );
false
```

### 2.3.2 Julia

▷ Julia (global variable)

This global variable represents the Julia module Main, see IsJuliaModule (2.1.4).

The variables from the underlying Julia session can be accessed via Julia, as follows.

Example

```
gap> Julia.sqrt;  # a Julia function
<Julia: sqrt>
gap> JuliaEvalString( "x = 1" );  # an assignment in the Julia session
1
gap> Julia.x;  # access to the value that was just assigned
1
gap> Julia.Main.x;
1
```

### 2.3.3 ImportJuliaModuleIntoGAP

▷ `ImportJuliaModuleIntoGAP(`*`name`*`)` (function)

**Returns:** nothing.

The aim of this function is to make the Julia module with name *`name`* available in the current GAP session. After the call, the *`name`* component of the global object `Julia` (2.3.2) will be bound, and one can access the module as a component of `Julia` (2.3.2) or via `JuliaModule` (2.3.4).

Example

```
gap> ImportJuliaModuleIntoGAP( "GAP" );
gap> Julia.GAP;
<Julia module GAP>
```

The Julia modules `Base`, `Core`, and `GAP` have in fact already been imported when the JuliaInterface package got loaded.

### 2.3.4 JuliaModule

▷ `JuliaModule(`*`name`*`)` (function)

**Returns:** a Julia object

Returns the Julia object that points to the Julia module with name *`name`*. Note that this module needs to be imported before being present, see `ImportJuliaModuleIntoGAP` (2.3.3).

Example

```
gap> gapmodule:= JuliaModule( "GAP" );
<Julia: GAP>
gap> gapmodule = JuliaPointer( Julia.GAP );
true
```

### 2.3.5 JuliaTypeInfo

▷ `JuliaTypeInfo(`*`juliaobj`*`)` (function)

**Returns:** a string.

Returns the string that describes the Julia type of the object *`juliaobj`*.

Example

```
gap> JuliaTypeInfo( Julia.GAP );
"Module"
gap> JuliaTypeInfo( JuliaPointer( Julia.GAP ) );
"Module"
gap> JuliaTypeInfo( JuliaEvalString( "sqrt(2)" ) );
"Float64"
gap> JuliaTypeInfo( 1 );
"Int64"
```

### 2.3.6 CallJuliaFunctionWithCatch

▷ `CallJuliaFunctionWithCatch(`*`juliafunc, arguments`*`)` (function)

**Returns:** a record.

The function calls the Julia function *`juliafunc`* with arguments in the GAP list *`arguments`*, and returns a record with the components `ok` and `value`. If no error occured then `ok` has the value `true`, and `value` is the value returned by *`juliafunc`*. If an error occured then `ok` has the value `false`, and `value` is the error message as a GAP string.

Example

```
gap> fun:= Julia.sqrt;;
gap> CallJuliaFunctionWithCatch( fun, [ 2 ] );
rec( ok := true, value := <Julia: 1.4142135623730951> )
gap> res:= CallJuliaFunctionWithCatch( fun, [ -1 ] );;
gap> res.ok;
false
gap> res.value{ [ 1 .. Position( res.value, '(' )-1 ] };
"DomainError"
gap> inv:= Julia.inv;;
gap> m:= GAPToJulia( JuliaEvalString( "Array{Int,2}" ), [[1,2],[2,4]] );
<Julia: [1 2; 2 4]>
gap> res:= CallJuliaFunctionWithCatch( inv, [ m ] );;
gap> res.ok;
false
gap> res.value{ [ 1 .. Position( res.value, '(' )-1 ] };
"LinearAlgebra.SingularException"
```

### 2.3.7 CallJuliaFunctionWithKeywordArguments

▷ CallJuliaFunctionWithKeywordArguments(*juliafunc, arguments, arec*) (function)

**Returns:** the result of the Julia function call.

The function calls the Julia function *juliafunc* with ordinary arguments in the GAP list *arguments* and keyword arguments given by the component names (keys) and values of the record *arec*, and returns the function value.

Note that the entries of *arguments* and the components of *arec* are not implicitly converted to Julia.

Example

```
gap> CallJuliaFunctionWithKeywordArguments( Julia.Base.round,
>        [ GAPToJulia( Float( 1/3 ) ) ], rec( digits:= 5 ) );
<Julia: 0.33333>
gap> CallJuliaFunctionWithKeywordArguments(
>        Julia.Base.range, [ 2 ], rec( length:= 5, step:= 2 ) );
<Julia: 2:2:10>
gap> m:= GAPToJulia( JuliaEvalString( "Array{Int,2}" ),
>            [ [ 1, 2 ], [ 3, 4 ] ] );
<Julia: [1 2; 3 4]>
gap> CallJuliaFunctionWithKeywordArguments(
>        Julia.Base.reverse, [ m ], rec( dims:= 1 ) );
<Julia: [3 4; 1 2]>
gap> CallJuliaFunctionWithKeywordArguments(
>        Julia.Base.reverse, [ m ], rec( dims:= 2 ) );
<Julia: [2 1; 4 3]>
gap> tuptyp:= JuliaEvalString( "Tuple{Int,Int}" );;
gap> t1:= GAPToJulia( tuptyp, [ 2, 1 ] );
<Julia: (2, 1)>
gap> t2:= GAPToJulia( tuptyp, [ 1, 3 ] );
<Julia: (1, 3)>
gap> CallJuliaFunctionWithKeywordArguments(
>        Julia.Base.( "repeat" ), [ m ],
>        rec( inner:= t1, outer:= t2 ) );
```

```
<Julia: [1 2 1 2 1 2; 1 2 1 2 1 2; 3 4 3 4 3 4; 3 4 3 4 3 4]>
```

## 2.4 Calling Julia functions

The simplest way to execute Julia code from GAP is to call `JuliaEvalString` (2.2.1) with a string that contains the Julia code in question.

Example

```
gap> JuliaEvalString( "sqrt( 2 )" );
<Julia: 1.4142135623730951>
```

However, it is usually more suitable to create GAP variables whose values are Julia objects, and to call Julia functions directly. The GAP function call syntax is used for that.

Example

```
gap> jsqrt:= JuliaEvalString( "sqrt" );
<Julia: sqrt>
gap> jsqrt( 2 );
<Julia: 1.4142135623730951>
```

In fact, there are slightly different kinds of function calls. A Julia function such as `Julia.sqrt` (or equivalently `JuliaFunction( "sqrt" )`) is represented by a GAP function object, and calls to it are executed on the C level, using Julia's `jl_call`.

Example

```
gap> fun:= Julia.sqrt;
<Julia: sqrt>
gap> IsJuliaObject( fun );
false
gap> IsFunction( fun );
true
gap> fun( 2 );
<Julia: 1.4142135623730951>
```

There are also callable Julia objects which aren't represented by GAP functions, for example Julia types can be called like functions. In this situation, the function call is executed via the applicable `CallFuncList` (**Reference: CallFuncList**) method, which calls Julia's `Core._apply`.

Example

```
gap> smalltype:= Julia.Int32;
<Julia: Int32>
gap> IsJuliaObject( smalltype );
true
gap> IsFunction( smalltype );
false
gap> val:= smalltype( 1 );
<Julia: 1>
gap> JuliaTypeInfo( val );
"Int32"
gap> JuliaTypeInfo( 1 );
"Int64"
```

### 2.4.1 Convenience methods for Julia objects

For the following operations, methods are installed that require arguments in `IsJuliaObject` (2.1.1) and delegate to the corresponding Julia functions.

- `CallFuncList` (**Reference: CallFuncList**), delegating to `Julia.Core._apply` (this yields the function call syntax in GAP, it is installed also for objects in `IsJuliaWrapper` (2.1.2),
- access to and assignment of entries of arrays, via `\[\]` (**Reference: \[\]**), `\[\]\:\=` (**Reference: \[\]\:\=**), and the (up to GAP 4.11 undocumented) operations `MatElm` and `SetMatElm`, delegating to `Julia.Base.getindex` and `Julia.Base.setindex`,
- access to and assignment of fields and properties, via `\.` (**Reference: Title page**) and `\.\:\=` (**Reference: \.\:\=**), delegating to `Julia.Base.getproperty` and `Julia.Base.setproperty`,
- the unary arithmetic operations `AdditiveInverseOp` (**Reference: AdditiveInverseOp**), `ZeroOp` (**Reference: ZeroOp**), and `OneOp` (**Reference: OneOp**), delegating to `Julia.Base.\-`, `Julia.Base.zero`, and `Julia.Base.one`,
- the binary arithmetic operations `\+` (**Reference: +**), `\-` (**Reference: -**), `\*` (**Reference: *), `\/` (**Reference: /**), `LeftQuotient` (**Reference: LeftQuotient**), `\^` (**Reference: ^**), `\=` (**Reference: \=**), `\<` (**Reference: \<**), delegating to `Julia.Base.\+`, `Julia.Base.\-`, `Julia.Base.\*`, `Julia.Base.\/`, `Julia.Base.\\`, `Julia.Base.\^`, `Julia.Base.\=\=`, and `Julia.Base.\<`; the same methods are installed also for the case that only one argument is in `IsJuliaObject` (2.1.1), and the other argument is an immediate integer.

Example

```
gap> m:= GAPToJulia( JuliaEvalString( "Array{Int,2}" ),
>            [ [ 1, 2 ], [ 3, 4 ] ] );
<Julia: [1 2; 3 4]>
gap> m[1,2];
2
gap> - m;
<Julia: [-1 -2; -3 -4]>
gap> m + m;
<Julia: [2 4; 6 8]>
```

## 2.5 Access Julia help from a GAP session

In a Julia session, one can ask for help about the object with the name `obj` (a function or a type) by entering `?obj`, and Julia prints all matches to the screen. One can get the same output in a GAP session by entering `?Julia:obj`, cf. Section (**Reference: Invoking the Help**) in the GAP Reference Manual. For example, `?Julia:sqrt` shows the Julia help about the Julia function `sqrt` (which is available in GAP as `Julia.sqrt`).

Note that this way to access the Julia help is different from the usual access to GAP help books, in the following sense.

- The qualifying prefix `Julia:` is mandatory. Thus the help request `?sqrt` will show matches from usual GAP help books (there is one match in the GAP Reference Manual), but not the help abot the Julia function `sqrt`.

- Since the prefix `Julia:` does not belong to a "preprocessed" help book with chapters, sections, index, etc., help requests of the kinds `?<`, `?<<`, `?>`, `?>>` are not meaningful when the previous help request had the prefix `?Julia:`. (Also requests with the prefix `??Julia:` do not work, but this holds also for usual GAP help books.)
- The Julia help system is case sensitive. Thus `?Julia:sqrt` yields a match but `?Julia:Sqrt` does not, and `?Julia:Set` yields a match but `?Julia:set` does not.
- The Julia help system does currently not support menus in case of multiple matches, all matches are shown at once, and this happens also in a GAP session.
- No PDF or HTML version of the Julia help is supported in GAP, only the text format can be shown on the screen. Thus it does not make sense to change the help viewer, cf. Section (**Reference: Changing the Help Viewer**) in the GAP Reference Manual.
- Julia functions belong to Julia modules. Many Julia functions can be accessed only relative to their modules, and then also the help requests work only for the qualified names. For example, `?Julia:GAP.julia_to_gap` yields the description of the Julia function `julia_to_gap` that is defined in the Julia module `GAP`, whereas no match is found for the input `?Julia:julia_to_gap`.

# Chapter 3

# Conversions between GAP and Julia

## 3.1 Conversion rules

### 3.1.1 Guiding principles

**Avoid conversions, use wrapper objects instead.**

Naively, one may think that in order to use Julia functionality from GAP, one has to convert all data to a format usable by Julia, then call Julia functions on that data, and finally convert it back; rinse and repeat. While this is certainly sometimes so, in many cases, things are a bit different: Some initial (usually very small) data may need to be converted. But afterwards, the output of one Julia function is used as input of the next one, and so on. Converting the data to GAP format and back then is needlessly wasteful. It is much better to not perform any conversion here. Instead, we create special "wrapper" objects on the GAP side, which wraps a given Julia object without converting it. This operation is thus very cheap, both in terms of performance and in memory usage. Such a wrapped object can then be transparently used as input for Julia functions.

On the GAP C kernel level, the internal functions used for this are `NewJuliaObj`, `IS_JULIA_OBJ`, `GET_JULIA_OBJ`. On the GAP language level, this is `IsJuliaObject` (2.1.1). On the Julia side, there is usually no need for a wrapper, as (thanks to the shared garbage collector) most GAP objects are valid Julia objects of type `GAP.GapObj`. The exception to that rule are immediate GAP objects, more on that in the next section.

**Perform automatic conversions only if absolutely necessary, or if unambiguous and free.**

Any conversion which the user cannot prevent, and which has some cost or choice involved, may cause several problems. The added overhead may turn an otherwise reasonable computation into an infeasible one (think about a conversion triggered several million times). And the conversion can add extra complications if one wants to detect and undo it.

**Provide explicit conversion functions for as many data types as possible.**

While users should not be forced into conversions, it nevertheless should be possible to perform sensible conversions. The simpler it is to do so, the easier it is to use the interface.

**Conversion round trip fidelity.**

If an object is converted from Julia to GAP and back to Julia (or conversely, from GAP to Julia and back to GAP), then ideally the result should be equal and of equal type to the original

value. At the very least, the automatic conversions should follow this principle. This is not always possible, due to mismatches in existing types, but we strive to get as close as possible.

### 3.1.2 Automatic (implicit) conversions

GAP has a notion of "immediate" objects, whose values are stored inside the "pointer" referencing them. GAP uses this to store small integers and elements of small finite fields, see for example the beginning of Chapter (**Reference: Integers**) in the GAP Reference Manual. Since these are not valid pointers, Julia cannot treat them like other GAP objects, which are simply Julia objects of type `GAP.GapObj`. Instead, a conversion is unavoidable, at least when immediate objects are passed as stand-alone arguments to a function.

To this end, the interface converts GAP immediate integers into Julia `Int64` objects, and vice versa. However, GAP immediate integers on a 64 bit system can only store 61 bits, so not all `Int64`objects can be converted into immediate integers; integers exceeding the 61 bits limit are therefore wrapped like any other Julia object. Other Julia integer types, like `UInt64`, `Int32`, are also wrapped by default, in order to ensure that conversion round trips do not arbitrary change object types.

All automatic conversions and wrappings are handled on the C functions `julia_gap` and `gap_julia` in JuliaInterface.

The following conversions are performed by `julia_gap` (from GAP's `Obj` to Julia's `jl_value_t*`).

- `NULL` to `jl_nothing`,
- immediate integer to `Int64`,
- immediate FFE to the `GapFFE` Julia type,
- GAP `true` to Julia `true`,
- GAP `false` to Julia `false`,
- Julia object wrapper to Julia object,
- Julia function wrapper to Julia function,
- other GAP objects to `GAP.GapObj`.

The following conversions are performed by `gap_julia` (from Julia's `jl_value_t*` to GAP's `Obj`).

- `Int64` to immediate integer when it fits, otherwise to a GAP large integer,
- `GapFFE` to immediate FFE,
- Julia `true` to GAP `true`,
- Julia `false` to GAP `false`,
- `GAP.GapObj` to `Obj`,
- other Julia objects to Julia object wrapper.

### 3.1.3 Manual (explicit) conversions

Manual conversion in GAP is done via the functions `GAPToJulia` (3.2.2) and `JuliaToGAP` (3.2.1). In Julia, conversion is done via `gap_to_julia` and `julia_to_gap`.

*Conversion from GAP to Julia*

In GAP, the function `GAPToJulia` (3.2.2) calls (after automatic conversion of the GAP object if applicable) the Julia function `gap_to_julia`; If a Julia type has been entered as the first argument of `GAPToJulia` (3.2.2) then this is the type to which the GAP object shall be converted, and if such a conversion is implemented then a Julia object of this type is returned, otherwise an `ArgumentError` is thrown.

- `IsBool` to `Bool`,
- `IsFFE and IsInternalRep` to `GapFFE`,
- `IsInt and IsSmallIntRep` to `Int8`, `Int16`, `Int32`, `Int64` (default), `Int128`, `UInt8`, `UInt16`, `UInt32`, `UInt64`, `UInt128`, `BigInt`, `Rational{T} where T <: Integer`,
- `MPtr and IsInt` to `BigInt` (default), `Rational{T} where T <: Integer`,
- `MPtr and IsRat` to `Rational{BigInt}` (default), `Rational{T} where T <: Integer`,
- `IsFloat` to `Float16`, `Float32`, `Float64` (default), `BigFloat`,
- `IsChar` to `Cuchar`,
- `IsString` to `AbstractString` (default), `String`, `Symbol`, `Array{UInt8,1}`, or types available for `IsList`,
- `IsRange` to `StepRange{Int64,Int64}` (default), `StepRange{T1,T2}`, `UnitRange{T}`, or types available for `IsList`,
- `IsBlist` to `BitArray{1}` (default), or types available for `IsList`,
- `IsList` to `Array{Union{Any,Nothing},1}` (default), `Array{T,1}`, `Array{T,2}`, `T <: Tuple`,
- `IsRecord` to `Dict{Symbol,Any}` (default), `Dict{Symbol,T}`.

If no Julia type is specified then a Julia type is chosen, based on the filters of the GAP object, see the "(default)" markers in the above list. Note that this might include checking various filters and will be, in almost all cases, slower than the typed version.

*Conversion from Julia to GAP*

The function `JuliaToGAP` (3.2.1) is a GAP constructor taking two arguments, a GAP filter and an object to be converted. Various methods for this constructor then take care of input validation and the actual conversion, either by delegating to the Julia function `julia_to_gap` (which takes just one argument and chooses the GAP filters of its result depending on the Julia type), or by automatic conversion. The supported Julia types are as follows.

| Julia type | GAP filter | comment |
|---|---|---|
| Int64, MPtr, GapFFE, and Bool | | automatic conversion |
| other integers, including BigInt | IsInt | integers |
| Rational{T} | IsRat | rationals |
| Float16, Float32, Float64 | IsFloat | machine floats |
| AbstractString | IsString | strings |
| Symbol | IsString | strings |
| Array{T,1} | IsList | plain lists |
| Array{Bool,1}, BitArray{1} | IsBList | bit lists |
| Tuple{T} | IsList | plain lists |
| Dict{String,T}, Dict{Symbol,T} | IsRecord | records |
| UnitRange{T}, StepRange{T} | IsRange | ranges |

## 3.2 Conversion functions

### 3.2.1 JuliaToGAP (for IsObject, IsObject)

▷ JuliaToGAP(*filt*, *juliaobj*[, *recursive*]) (constructor)

**Returns:** a GAP object in the filter *filt*

Let *juliaobj* be an object in IsArgumentForJuliaFunction (2.1.5) for which a conversion to GAP is provided, in the sense of Section 3.1, such that the corresponding GAP object is in the filter *filt*. Then JuliaToGAP returns this GAP object.

For recursive structures (GAP lists and records), only the outermost level is converted except if the optional argument *recursive* is given and has the value true, in this case all layers are converted recursively.

Note that this default is different from the default in the other direction (see GAPToJulia (3.2.2)). The idea behind this choice is that from the viewpoint of a GAP session, it is more likely to use plain Julia objects for computations on the Julia side than Julia objects that contain GAP subobjects, whereas "shallow conversion" of Julia objects to GAP yields something useful on the GAP side.

Example

```
gap> s:= GAPToJulia( "abc" );
<Julia: "abc">
gap> JuliaToGAP( IsString, s );
"abc"
gap> l:= GAPToJulia( [ 1, 2, 4 ] );
<Julia: Any[1, 2, 4]>
gap> JuliaToGAP( IsList, l );
[ 1, 2, 4 ]
```

The following values for *filt* are supported. IsInt (**Reference: IsInt**), IsRat (**Reference: IsRat**), IsFFE (**Reference: IsFFE**), IsFloat (see (**Reference: Floats**)), IsBool (**Reference: IsBool**), IsChar (**Reference: IsChar**), IsRecord (**Reference: IsRecord**), IsString (**Reference: IsString**), IsRange (**Reference: IsRange**), IsBlist (**Reference: IsBlist**), IsList (**Reference: IsList**). See Section 3.1 for the admissible types of *juliaobj* in these cases.

### 3.2.2 GAPToJulia

▷ GAPToJulia( *[juliatype, ]gapobj[, recursive]* ) (function)

**Returns:** a Julia object

Let *gapobj* be an object for which a conversion to Julia is provided, in the sense of Section 3.1, such that a corresponding Julia object with type *juliatype* can be constructed. Then GAPToJulia returns this Julia object.

If *juliatype* is not given then a default type is chosen. The function is implemented via the Julia function GAP.gap_to_julia.

Example

```
gap> GAPToJulia( 1 );
1
gap> GAPToJulia( JuliaEvalString( "Rational{Int64}" ), 1 );
<Julia: 1//1>
gap> l:= [ 1, 3, 4 ];;
gap> GAPToJulia( l );
<Julia: Any[1, 3, 4]>
gap> GAPToJulia( JuliaEvalString( "Array{Int,1}" ), l );
<Julia: [1, 3, 4]>
gap> m:= [ [ 1, 2 ], [ 3, 4 ] ];;
gap> GAPToJulia( m );
<Julia: Any[Any[1, 2], Any[3, 4]]>
gap> GAPToJulia( JuliaEvalString( "Array{Int,2}" ), m );
<Julia: [1 2; 3 4]>
gap> r:= rec( a:= 1, b:= [ 1, 2, 3 ] );;
gap> GAPToJulia( r );
<Julia: Dict{Symbol,Any}(:a => 1,:b => Any[1, 2, 3])>
```

If *gapobj* is a list or a record, one may want that its subobjects are also converted to Julia or that they are kept as they are, which can be decided by entering true or false as the value of the optional argument *recursive*; the default is true, that is, the subobjects of *gapobj* are converted recursively.

Note that this default is different from the default in the other direction, see the description of JuliaToGAP (3.2.1).

Example

```
gap> jl:= GAPToJulia( m, false );
<Julia: Any[GAP: [ 1, 2 ], GAP: [ 3, 4 ]]>
gap> jl[1];
[ 1, 2 ]
gap> jr:= GAPToJulia( r, false );
<Julia: Dict{Symbol,Any}(:a => 1,:b => GAP: [ 1, 2, 3 ])>
gap> Julia.Base.get( jr, JuliaSymbol( "b" ), fail );
[ 1, 2, 3 ]
```

## 3.3 Open items

- Discuss/add more dedicated conversion functions and/or special wrapper kinds, e.g.:
  - There could be a Julia type hierarchy of wrappers, e.g., GAPInt <: GAPRat <: GAPCyc; those types would wrap the corresponding GAP objects, i.e., they would simply wrap a Union{MPtr,Int64}, but perhaps provided nicer integration with the rest of

Julia, like methods for gcd, say, which are properly type restricted; or nicer printing (w/o the GAP: prefix even?). Not really sure whether this is useful, though.

- Should we allow the three argument case of JuliaToGAP (3.2.1) in all cases, e.g., JuliaToGAP( IsInt, 1, true )?
- Many tests of conversions are missing.

# Index