REPORT ON OpenDreamKit DELIVERABLE D5.15

Final report and evaluation of all the GAP developments.

ALEXANDER KONOVALOV, STEPHEN LINTON AND MICHAEL TORPEY

Due on	31/08/2019 (M48)
Delivered on	31/08/2019
Lead	University of St Andrews (USTAN)
Progress on and finalization of this deliverable has been tracked publicly at: `https://github.com/OpenDreamKit/OpenDreamKit/issues/113`	

DELIVERABLE DESCRIPTION, AS TAKEN FROM GITHUB ISSUE #113 ON 2019-08-31

- **WP5:** High Performance Mathematical Computing
- **Lead Institution:** University of St Andrews
- **Due:** 2019-08-31 (month 48)
- **Nature:** Report
- **Task:** T5.2 (#100)
- **Proposal:** p. 51
- **Final report** (sources)

GAP ("Groups, Algorithms, Programming") is a software system for computational discrete mathematics, with its origins in finite group theory. The system has been developed continually since 1986, with the first public release in 1988. Throughout its life it has been made available free of charge for use, extension and non-commercial redistribution. It is widely used by researchers in mathematics, computer science and other disciplines, and has been cited in more than 3000 research publications. It is also widely used in teaching. GAP is also used as a component of other mathematical software systems such as SageMath and OSCAR.

This report describes the work undertaken in connection with Task 5.2: a wide range of developments within the GAP system to meet the general objective of workpackage 5 ("improve the performance of the computational components of OpenDreamKit"). We have worked on supporting the use of a range of High Performance Computing environments, prioritizing those most accessible to our users, namely multi-core systems, adding to our existing support for clusters and cloud computing. We have also made fundamental performance improvements and improvements to our tools, processes and to the programming language, that benefit all GAP users.

A collection of GAP Jupyter notebooks, together with the setup to execute them on Binder, is available at `https://github.com/OpenDreamKit/gap-demos`.

Acknowledgement: The authors listed above are the authors of this report. The work reported on includes contributions from many others, both within and beyond the OpenDreamKit project. Authorship of individual software contributions can generally be identified from the documentation and/or the version control records of the systems concerned.

CONTENTS

1. Introduction

This report describes the work undertaken in connection with Task 5.2: a wide range of developments within the GAP system to meet the general objective of workpackage 5 ("improve the performance of the computational components of OpenDreamKit"). We have worked on supporting the use of a range of High Performance Computing environments, prioritizing those most accessible to our users, namely multi-core systems, adding to our existing support for clusters and cloud computing. We have also made fundamental performance improvements and improvements to our tools, processes and to the programming language, that benefit all GAP users.

In a system such as GAP, which is actively developed by a widely distributed team, only a few of whom are part of the OpenDreamKit project, any developments which are to be useful to users of GAP (whether as a component of OpenDreamKit or not) need to be maintainable, and integrate well with the general flow of GAP development, which itself has to be nurtured and supported. Because of this, it is neither possible nor desirable to clearly delineate every development which is part of Task 5.2 from every one which is not. On the other hand, the impetus and resources from the OpenDreamKit project have meant that OpenDreamKit goals have pervaded a great deal of GAP's evolution over the last four years.

In light of this, this report will necessarily include a broad survey of all the major developments in GAP and its package ecosystem over the period, but we will concentrate on, and highlight, links between these developments and OpenDreamKit goals. Work which is specifically part of other work packages is reported in appropriate deliverables, but these are mainly quite focused, leaving a broad range of material for this report.

1.1. Context and Background

Recall that GAP ("Groups, Algorithms, Programming") is a software system for computational discrete mathematics, with its origins in finite group theory. The system has been developed continually since 1986, with the first public release in 1988. Throughout its life it has been made available free of charge for use, extension and non-commercial redistribution. It is widely used by researchers in mathematics, computer science and other disciplines, and has been cited in more than 3000 research publications. It is also widely used in teaching.

Thus far, GAP has normally been used via a single-threaded text-based "read–eval–print loop" user interface, with the same GAP language serving both for the interface and for the implementation of much of the system. A research user may use the system purely interactively, as a "desk calculator", but typically will extend the system with a small or large collection of GAP functions and complete their research calculations by interacting with those. One of the goals of GAP-related work in OpenDreamKit is to offer a range of flexible alternatives to this in line with current practices in other areas, and more suited for modern high-performance computers.

A key feature of GAP today is the rapidly growing ecosystem of contributed extensions called "packages" (145 are redistributed with GAP 4.10.2). These packages may: extend mathematical functionality of the system by implementing new algorithms; provide mathematical datasets, together with search and access functions, or provide enhanced tools for users and/or developers. Packages can be published independently of the core GAP system, and each has a separate authorship. Packages that pass our checks (for example having documentation and tests and not causing regressions in the core system or clashes with other packages) can (at the author's request) be automatically redistributed with GAP releases. Again at the author's choice, they may be submitted for "refereeing", a process designed to be analogous to the refereeing of scientific papers, providing both quality control for the user and recognition for the author.

For the last decade or more, the majority of the development work in the GAP ecosystem has been in packages. Another area of GAP development under OpenDreamKit is to ensure that the

package system (including the infrastructure used to gather, test and release packages) scales as the number of packages grows.

Prior to the start of OpenDreamKit, a number of initiatives had developed alternative models for the use of GAP. SAGEMATH (SAGE hereafter) has used GAP for much of its group theoretic functionality since 2006. One benefit of that was that it allowed the use of GAP functionality through SAGE's notebook user interface. In the Framework 6 project "SCIEnce: Symbolic Computation Infrastructure for Europe" we developed a powerful and flexible remote procedure call interface to GAP using the OpenMath data representation format, called SCSCP (Symbolic Computation Software Composability Protocol) one of whose roles was to support coarse-grained distributed computation. In a UK-funded software development project, HPC-GAP, we began the development of an intrinsically multi-threaded version of GAP. As well as new software, these projects gave us insight into the parallel and high-performance computing needs of our user community, leading us, for instance, to remain more focused on multi-core servers and *ad hoc* clusters (such as a mathematical research group or department might provide) than on "supercomputers". Even on these systems, interactive use rather than "batch" computation remains the norm, and we aim to support that, and enhance it through Jupyter notebooks rather than force users to a model that does not suit them.

The central thrust of the involvement of GAP in OpenDreamKit is increased diversity of users and models of use. To this end, in WP3 we have added support for direct use of GAP via Jupyter notebooks as an alternative to our 1980s text-based user interface. In this work package we report the work intended to support GAP users in performing larger and more demanding computations. This includes general work on increasing the performance and robustness of GAP and its packages, continued development of our support for shared memory parallel computing via HPC-GAP and MEATAXE64 and maintenance of our support for distributed memory parallel computing using the SCSCP package [19]. We also report enhancements to the GAP language and to GAP's development and debugging tools and our support for package developers. More demanding computations require more complex software to be written, debugged, optimized and shared, and these changes enable that.

1.2. **Overview**

This report is divided into sections as follows: section 2 reports on two major initiatives supporting parallel computation on multi-core shared memory systems: HPC-GAP and MEATAXE64. Section 3 reports a wide range of relevant general developments in the core GAP system, highlighting LIBGAP which gives a much faster and more reliable link to other systems and the new linguistic reflection facilities which will provide a basis for future developments in optimization and automatic parallelization. Section 4 similarly surveys relevant developments among the many new or updated packages. Section 5 discusses our work aimed at improving robustness and reliability, both of GAP and its packages. This includes both new tool support and new processes.

2. High-Performance Computing with GAP

Experiments with parallel computing in GAP date back to the late 1990s. The ParGAP package by Gene Cooperman and others [11] implemented a limited set of MPI bindings and experimented with a number of variations of the master–worker paradigm. It was not widely adopted due, we believe, to a lack of suitable clusters available to typical GAP users at the time, and issues with robustness, installation and debugging support.

Within the Framework 6 project SCIEnce: Symbolic Computation Infrastructure for Europe, we developed the SCSCP package [19], which provided a much more robust remote procedure call architecture for GAP which still serves a variety of uses, including effective support for coarse-grained distributed memory parallel computation, suitable for a cloud or *ad hoc* cluster environment, as seen for example, in [21]. Compared to MPI, it is much less efficient, but much more fault-tolerant and more adaptable to heterogeneous environments.

In this project, we have therefore focused our attention on shared memory parallelisation, relevant to many of our users given that even modern laptops may have up to six cores, while 64 core servers or cloud nodes are widely available to many research mathematicians.

2.1. **HPC-GAP: Multi-Threaded Programming in GAP**

GAP 4.9.1 (Month 33) for the first time included experimental code to support safe multi-threaded programming in GAP, dubbed HPC-GAP. This had been in development for some time as a fork of GAP and the HPC-GAP and GAP codebases had diverged. In this release we reunified them, making thread support a compile-time option instead of requiring a completely different program. This was an essential step to support further development of HPC-GAP, and ensure that new developments in GAP could be quickly incorporated. It also provided general GAP users an opportunity to start to experiment with HPC-GAP, for which documentation is also provided.

HPC-GAP supports two models of parallel programming in GAP, which can be combined: threads and tasks. Threads operate as in most languages. The basic primitive is to start execution of a GAP function in a new thread, while the calling thread returns immediately. Threads can interact through global data, by passing interrupts or by returning results when they exit. Thread-local variables are available as an alternative to global variables.

Tasks are similar, but a task is submitted to a shared job queue and executed by one of a fixed-size pool of worker threads. Typically the calling program will start a number of tasks and then wait for some or all of them to complete, leaving the task scheduler to allocate each task to a worker. A task can be scheduled to run only when another has completed, and more complex dependencies can be expressed using "milestones".

A number of data structures and other features are provided to simplify the programming of interactions between tasks and/or threads. These were developed in consultation with a group of GAP developers, based on the specific types of computation that they anticipated needing to parallelize. They include:

- Atomic lists and records, which behave like standard GAP data structures, but guarantee that their basic access operations are thread-safe;
- Channels (efficient first-in-first-out data structures designed to pass objects between threads or tasks);
- Synchronization variables which guarantee that exactly one write to the variable will succeed; and
- Traditional semaphores.

Full details can be found in the HPC-GAP manual in Appendix A.

Implementing all of this efficiently required a substantial refactoring of the GAP interpreter, but presented no other challenges unique to GAP. What does present a serious challenge for HPC-GAP, however, is that the GAP library makes very extensive use of shared global data,

which is frequently updated as new knowledge is obtained. Removing this sharing of knowledge would make the overall system much less effective, as well as requiring millions of lines of code to be rewritten, but ensuring that access and updates to this data from different threads and tasks do not cause corruption or inconsistency was difficult.

The approach taken was to limit access, in most cases, to data "owned" by the accessing thread, read-only data and "public" data stored in thread-safe data structures which were used for the global "knowledge pool". In particular, this last set of data could be updated "atomically", so any thread would see the old state *or* the new one, but nothing in between. The access limitations are implemented by access checks in the kernel, with the aim of ensuring that most unsafe data access should result in error messages, rather than inconsistent results. Experiments with this system have been reported in a number of papers, for instance [4].

The main remaining challenge is to complete the adaptation of the enormous legacy codebase represented by GAP's library and key packages to use these mechanisms effectively, and to keep the underlying mechanisms working as the core GAP codebase evolves. Fusing the codebases, as was done in GAP 4.9, was a critical step in this work, meaning, for instance, that all changes to GAP are tested in HPC-GAP automatically.

The effectiveness of the underlying task mechanism is illustrated in the graph in Figure 1, which shows the speedups (wall-clock time on n cores versus CPU time on one core) achieved in running 1000 test tasks (each taking about 300ms) with various numbers of worker threads on a 64 core AMD "bulldozer" system. These cores share L2 cache and decode hardware in pairs, limiting performance with larger numbers of threads.

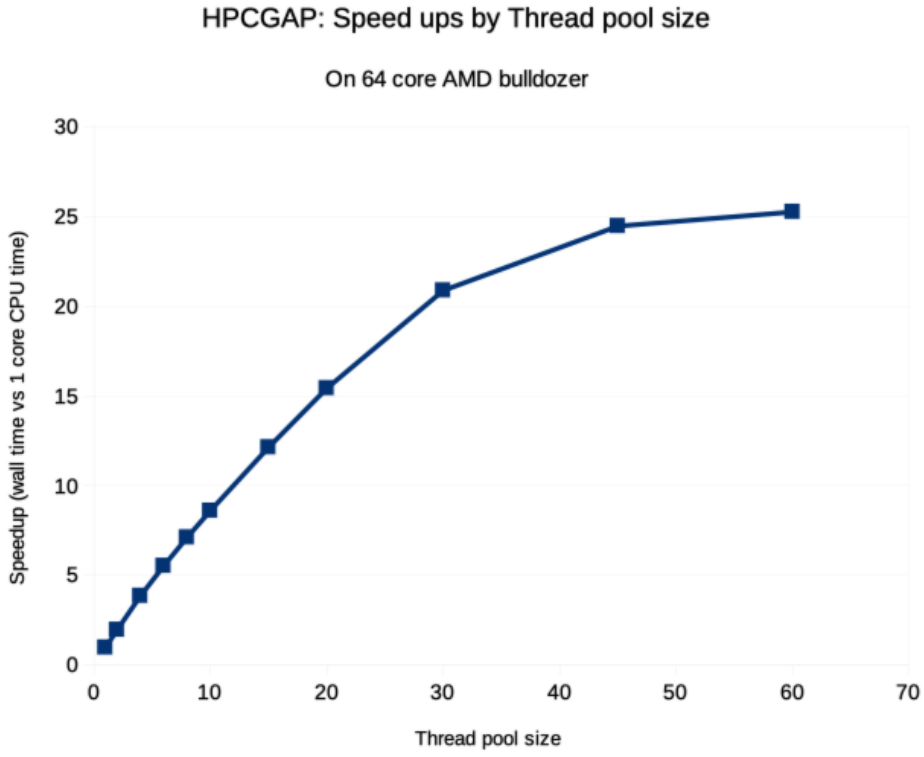


FIGURE 1. Speedups achieved by HPCGAP on 1000 test tasks

2.2. **MeatAxe64: High-Performance Linear Algebra over Finite Fields**

A key kernel for many applications of GAP is linear algebra for mainly dense matrices over a range of finite fields, especially small ones. This is used directly in working with matrix groups

and in computational representation theory, but also arises indirectly in areas such as graph theory, finite solvable groups and number theory.

The GAP kernel already includes memory-efficient representations of matrices over these fields, and implementations based on the C-MEATAXE (the name "MeatAxe" is derived from the abbreviation "mtx" for "matrix" and the fact that a key operation in computational representation theory is to "chop" a module into its irreducible components).

Working with external collaborators (primarily R. A. Parker), we have supported (and continue to contribute to) the design and development of a new C and assembler library, called MEATAXE64, which takes advantage of new algorithmic ideas, and of powerful features of modern CPUs, to deliver a quantum leap in performance for matrix multiplication and Gaussian elimination on modern multi-core shared memory computers. This project has involved radical innovations in low-level data formats and arithmetic algorithms, in cache-friendly loop structures (similar to those now standard in BLAS implementations, but made more complex by the need to reorganize and reformat both inputs and outputs of a multiplication), in matrix level reduction algorithms (both for non-prime fields of all characteristics, inspired by Albrecht [1], and along Strassen-Winograd lines), in a very efficient dataflow-driven thread farm and in high-level algorithms, especially for Gaussian elimination. A number of publications are in preparation.

We have implemented a GAP interface to this C library (the MEATAXE64 GAP package [20]) currently in beta-testing, and are in the process of revising higher levels of the software stack to make best use of it.

The MEATAXE64 C library is multi-threaded using a bespoke and effective thread-farm that supports a powerful dataflow module of computation. With this, the library can efficiently make use of multiple cores for large enough matrices, whether used from HPC-GAP or single-threaded GAP. Particular care is taken to limit the use of shared caches and of memory bandwidth in each task, so that they interfere as little as possible.

Tables 2, 3 and 4 indicate the performance gains and speedups obtained by MEATAXE64 compared to our existing code. The data is based on run-times for multiplication of two random dense square matrices of the given dimension with entries from $GF(q)$. In these tables the throughput is calculated as the cube of the dimension divided by the elapsed ("wall-clock") time, ignoring algorithmic techniques that may reduce the actual number of field operations needed. The unit (Gfop/s) is billions of field operations per second.

Table 2: Performance of existing GAP code

q	Dim.	cpu (s)	wall (s)	throughput (Gfop/s)
2	800	0.004	0.003	149.6
2	1600	0.016	0.016	255.8
2	4000	0.159	0.159	401.7
2	8000	1.581	1.582	323.6
2	16000	14.241	14.280	286.8
2	40000	193.251	193.524	330.7
3	500	0.053	0.052	2.4
3	1000	0.364	0.364	2.8
3	2500	5.268	5.268	3.0
3	5000	41.529	41.533	3.0
3	10000	327.798	328.121	3.0
3	25000	5020.887	5025.466	3.1
4	400	0.017	0.017	3.8
4	800	0.112	0.112	4.6
4	2000	1.542	1.542	5.2
4	4000	12.107	12.108	5.3

Table 2: Performance of existing GAP code

q	Dim.	cpu (s)	wall (s)	throughput (Gfop/s)
4	8000	99.259	99.361	5.2
4	20000	1513.64	1515.528	5.3
5	300	0.021	0.021	1.3
5	600	0.147	0.148	1.5
5	1500	2.166	2.166	1.6
5	3000	17.106	17.108	1.6
5	6000	136.714	136.840	1.6
5	15000	2088.063	2089.934	1.6
7	200	0.009	0.009	0.9
7	400	0.063	0.062	1.0
7	1000	0.906	0.906	1.1
7	2000	7.32	7.321	1.1
7	4000	56.659	56.710	1.1
7	10000	875.08	875.901	1.1
16	200	0.007	0.007	1.2
16	400	0.044	0.044	1.4
16	1000	0.644	0.643	1.6
16	2000	4.978	4.978	1.6
16	4000	39.757	39.798	1.6
16	10000	628.858	629.651	1.6
17	100	0.003	0.003	0.4
17	200	0.018	0.018	0.4
17	500	0.273	0.273	0.5
17	1000	2.153	2.154	0.5
17	2000	17.119	17.131	0.5
17	5000	268.032	268.267	0.5
64	100	0.002	0.002	0.5
64	200	0.012	0.013	0.6
64	500	0.186	0.186	0.7
64	1000	1.448	1.448	0.7
64	2000	11.548	11.558	0.7
64	5000	294.897	297.453	0.4
81	100	0.003	0.003	0.4
81	200	0.019	0.019	0.4
81	500	0.284	0.284	0.4
81	1000	2.215	2.215	0.5
81	2000	21.285	21.353	0.4
81	5000	276.045	276.503	0.5
101	100	0.003	0.003	0.3
101	200	0.021	0.020	0.4
101	500	0.295	0.295	0.4
101	1000	2.287	2.287	0.4
101	2000	18.212	18.233	0.4
101	5000	321.134	322.902	0.4

Table 3: Performance of MEATAXE64 on one core

q	Dim.	cpu (s)	wall (s)	throughput (Gfop/s)
2	8000	0.478	0.490	1045.1
2	16000	3.476	3.541	1156.7
2	40000	47.125	47.713	1341.4
2	80000	327.569	331.744	1543.4
2	160000	3460.336	3522.279	1162.9
3	5000	0.566	0.578	216.3
3	10000	4.125	4.152	240.9
3	25000	56.788	56.971	274.3
3	50000	399.282	400.549	312.1
3	100000	2774.011	2798.002	357.4
4	4000	0.205	0.216	295.8
4	8000	1.475	1.480	346.0
4	20000	19.889	20.070	398.6
4	40000	140.18	141.497	452.3
4	80000	980.936	985.148	519.7
5	3000	0.585	0.588	45.9
5	6000	4.102	4.107	52.6
5	15000	53.351	53.394	63.2
5	30000	362.048	362.369	74.5
5	60000	2585.863	2589.959	83.4
7	2000	0.217	0.221	36.2
7	4000	1.541	1.545	41.4
7	10000	20.652	20.675	48.4
7	20000	145.626	145.854	54.8
7	40000	1014.896	1016.415	63.0
16	2000	0.085	0.089	89.7
16	4000	0.603	0.623	102.7
16	10000	8.538	8.625	115.9
16	20000	60.979	61.521	130.0
16	40000	426.445	430.764	148.6
17	1000	0.074	0.074	13.5
17	2000	0.51	0.510	15.7
17	5000	7.259	7.265	17.2
17	10000	50.043	50.081	20.0
17	20000	352.604	353.215	22.6
64	1000	0.034	0.038	26.0
64	2000	0.201	0.206	38.7
64	5000	2.57	2.604	48.0
64	10000	17.858	18.125	55.2
64	20000	125.306	127.344	62.8
81	1000	0.095	0.099	10.1
81	2000	0.492	0.497	16.1
81	5000	5.519	5.560	22.5
81	10000	39.363	39.592	25.3
81	20000	274.394	276.197	29.0
101	1000	0.107	0.108	9.3
101	2000	0.75	0.751	10.7
101	5000	10.374	10.385	12.0
101	10000	72.007	72.056	13.9
101	20000	507.043	507.795	15.8

Table 4: Performance of MEATAXE64 on 64 cores

q	Dim.	cpu (s)	wall (s)	throughput (Gfop/s)
2	80000	904.369	21.638	23662.2
2	160000	4638.836	151.735	26994.4
2	320000	33340.649	1040.551	31491.0
3	50000	1004.631	21.017	5947.6
3	100000	5443.318	196.317	5093.8
3	200000	39283.76	1201.458	6658.6
4	40000	355.516	10.157	6300.9
4	80000	2548.595	51.267	9987.0
4	160000	13560.306	538.945	7600.0
5	30000	974.333	18.874	1430.6
5	60000	6273.661	142.248	1518.5
5	120000	38362.156	1171.026	1475.6
7	20000	437.489	8.740	915.3
7	40000	3035.705	53.539	1195.4
7	80000	15050.673	465.735	1099.3
16	20000	89.048	7.138	1120.8
16	40000	1044.91	22.805	2806.4
16	80000	6212.503	202.530	2528.0
17	10000	141.266	2.999	333.4
17	20000	1001.397	17.793	449.6
17	40000	7038.438	123.414	518.6
64	10000	58.099	1.807	553.5
64	20000	342.525	8.892	899.6
64	40000	2310.548	45.095	1419.2
81	10000	120.235	2.640	378.7
81	20000	734.844	13.992	571.8
81	40000	4913.959	88.109	726.4
101	10000	216.491	4.269	234.3
101	20000	1508.308	26.374	303.3
101	40000	10572.959	182.572	350.5

3. Developments in the Core GAP System

This Section highlights some of the most important changes to the core GAP system released during the project which contribute to OpenDreamKit goals. Other important developments are included in GAP packages and described in later sections. More detailed descriptions of each major and minor GAP release, with links to the corresponding source code changes on GitHub, are contained in the "GAP - Changes from Earlier Versions" document. This document is redistributed with GAP and is also available on the GAP web site at `https://www.gap-system.org/Manuals/doc/changes/chap0.html` and included as an Appendix B to this report.

This work includes many contributions from collaborators outside OpenDreamKit, whose authorship can be seen from the version control history.

3.1. libGAP: Allowing 3rd Party Code to Link GAP Efficiently as a Library

The initial connection between Sage and GAP was developed around 2006, and relied on running a full GAP process with which Sage communicated through UNIX pipes, creating GAP input as text and then parsing the text which was output from GAP. This enabled an initial connection, but introduced considerable overhead converting objects to and from text representations and passing them between processes, and considerable unreliability, since Sage was essentially using GAP through an interface designed for humans, rather than programs.

A better solution is to provide a C API enabling GAP to be integrated into the Sage process, and allowing a GAP session to be started, objects created, functions called and results retrieved, via C function calls. This is, however, not a mode of operation for which GAP was ever designed, and implementing it in a robust and flexible way required considerable adaptations to the system, essentially disentangling the computational engine from the user interface, while retaining, for example, the ability to dynamically load C code from GAP packages into the process that was using libGAP. Another complication is the need to make sure that the GAP memory manager does not delete objects which the calling process will want in the future.

Based on an initial version developed within the Sage team, a fully general-purpose and flexible version was released in GAP 4.10.0 (Month 39).

Subsequent releases have improved and will continue to improve the robustness of libGAP, and to extend its API with new functionality (For the detailed descriptions, see Chapter 2 "Changes between GAP 4.9 and GAP 4.10" of the "GAP - Changes from Earlier Versions" manual, provided in Appendix B). This has allowed Sage to drop its custom modifications to GAP and use an official, documented and regularly tested GAP interface instead, starting from SageMath 8.6 (Month 41).

3.2. Linguistic Reflection – Supporting Transformation and Optimisation of GAP Functions at Runtime

The GAP programming language, in which much of the system is written, is interpreted and dynamically typed (somewhat like the better-known Python). It has a unique "method selection" system ([6]) in which the type of an object reflects mathematical information learned about it at runtime; method selection uses the known types of all arguments in order to choose a method.

A compiler for the language (converting GAP code into C extensions to the GAP kernel) has been available for many years as part of GAP, but gives surprisingly little speedup on most code, and has not seen widespread adoption. The main reason for this is that objects' types must still be checked, and method selection decisions made, at run-time, even in compiled code.

Sage, implemented initially in Python, faced many of the same problems, and addressed them by adopting, and promoting development of, the Cython project, which extends the Python language with C-like types and constructions to allow more effective compilation.

In WP5 the GAP team have sought a natural and usable way to deliver similar benefits for the GAP language (bearing in mind the relative sizes of the GAP and Python user and developer communities). After some experiments and a great deal of discussion across the community, we have settled on "Linguistic reflection", combined with code annotations (pragmas) as an approach that offered a good return of performance and other benefits for the effort invested.

The key infrastructural tools for both of these have now been added to the GAP language and interpreter. The new `SyntaxTree` function "unpacks" the workspace representation of a GAP function, converting it into a data structure which can be easily manipulated by GAP, and converted back into a new executable GAP function when needed. Code annotations in the form of pragmas are retained through this process, so that the programmer can use them to guide in program transformation, and different software transformers can use them to communicate. All of this functionality is publicly available now in the development version of GAP on GitHub and will be included in GAP 4.11 (expected November 2019).

Future versions will build on this for optimization, OpenMP-style semi-automatic parallelization and (our original motivating application) to produce specialized versions of functions which reduce the amount of run-time method selection required, allowing them to be executed more quickly and compiled to C more effectively.

As a small example, consider the following code fragment:

```
foo := function(n, x, y)
    local z,i;
    z := 0;
    for i in [1..n] do
        #% x and y always integers
        z := z+Gcd(x,y);
    od;
    return z;
end;
```

The `#%` here denotes a pragma. This can be unpacked to produce the cumbersome, but easily processed data structure shown in Figure 5.

This data structure contains all of the necessary information for a simple GAP program to locate the pragma and, using the information in it, replace the general function `Gcd` by the specialized `GcdInt` statically. Doing this by hand for the moment, we can produce the session in Figure 6. This shows the full round trip from one executable GAP function `foo` to an optimised one `bar` which completes in approximately one fifth the time.

Note: It is important to understand that this is *not* a source code level transformation. Automated transformation tools built on this foundation do not need to parse or emit GAP syntax, and all questions of scope, including bindings in closures, are already resolved. A small planned extension will add profiling information to the syntax tree when available.

Other applications for this technology include automated parallelization (in combination with HPC-GAP) and traditional code optimizations.

While we would have preferred to have had more time to exploit these facilities within the project, it was important to achieve consensus around a solution that fit the needs of GAP and its community and will provide a basis for developments for years to come.

3.3. **Developments in GAP 4.8 (Month 8)**

GAP 4.8 was the first major release of GAP published after the start[1] of the OpenDreamKit project. In preparation for the planned integration of the HPC-GAP branch (supporting GAP language-level threading) into the core system, some language extensions were backported to core GAP at this stage. This helped to unify the codebases of GAP 4 and HPC-GAP by allowing HPC-GAP code to be read in a single-threaded GAP session. For example, the

[1] In the report we refer to months of the project, i.e. Month 1 is September 2015, while Month 48 is August 2019.

```
gap> t:= SyntaxTree(foo)!.tree;
rec( name := "foo", nams := [ "n", "x", "y", "z", "i" ], narg := 3,
  nloc := 2,
  stats :=
    rec(
      statements :=
        [
          rec( lvar := 4, rhs := rec( type := "EXPR_INT", value := 0 ),
              type := "STAT_ASS_LVAR" ),
          rec(
              body :=
                [
                  rec( type := "STAT_PRAGMA",
                      value := "% x and y always integers" ),
                  rec( lvar := 4,
                      rhs :=
                        rec(
                          left :=
                            rec( lvar := 4, type := "EXPR_REF_LVAR" ),
                          right :=
                            rec(
                              args :=
                                [
                                  rec( lvar := 2,
                                      type := "EXPR_REF_LVAR" ),
                                  rec( lvar := 3,
                                      type := "EXPR_REF_LVAR" ) ],
                              funcref :=
                                rec( gvar := "Gcd",
                                  type := "EXPR_REF_GVAR" ),
                              type := "EXPR_FUNCCALL_2ARGS" ),
                          type := "EXPR_SUM" ),
                      type := "STAT_ASS_LVAR" ) ],
              collection :=
                rec( first := rec( type := "EXPR_INT", value := 1 ),
                  last := rec( lvar := 1, type := "EXPR_REF_LVAR" ),
                  type := "EXPR_RANGE" ), type := "STAT_FOR_RANGE2",
              variable := rec( lvar := 5, type := "EXPR_REF_LVAR" ) ),
          rec( obj := rec( lvar := 4, type := "EXPR_REF_LVAR" ),
              type := "STAT_RETURN_OBJ" ) ], type := "STAT_SEQ_STAT3" )
    , type := "EXPR_FUNC", variadic := false )
```

FIGURE 5. A Full Syntax Tree Data Structure

`atomic`, `readonly` and `readwrite` keywords used for controlling access to shared data in HPC-GAP were added as "no-ops" to the regular GAP language.

In addition, this release also provided support for much enhanced profiling, essential to all our performance engineering work. Previously, GAP allowed tracking of the CPU time spent executing each function, but the new features in this release reduced this granularity to individual lines of GAP code. Related developments also made it easier to track time spent on individual lines of C code in the GAP kernel. A further related feature makes it very easy to measure "test coverage" – the proportion of the system that has actually been executed by a given set of tests. The PROFILING package [17] by Christopher Jefferson (USTAN) provided excellent tools for transforming these profiles into various human-readable forms. These features supported later work on enhancing both the performance and the reliability of GAP, through easier identification of hot-spots and easier measurement of test quality.

```
gap> foo := function(n, x, y)
>     local z,i;
>     z := 0;
>     for i in [1..n] do
>         #% x and y always integers
>         z := z+Gcd(x,y);
>     od;
>     return z;
> end;
function( n, x, y ) ... end
gap> t := SyntaxTree(foo)!.tree;;
gap> u := StructuralCopy(t);;
gap> u.stats.statements[2].body[1].value := "% Gcd specialised to GcdInt";;
gap> u.stats.statements[2].body[2].rhs.right.funcref.gvar := "GcdInt";;
gap> bar := SYNTAX_TREE_CODE(u);;
gap> Print(bar);
function ( n, x, y )
    local z, i;
    z := 0;
    for i in [ 1 .. n ] do
        #% Gcd specialised to GcdInt
        z := z + GcdInt( x, y );
    od;
    return z;
end
gap> foo(10^7, 102334155, 165580141);; time
2245
gap> bar(10^7, 102334155, 165580141);; time;
424
```

FIGURE 6. An example of manual optimisation using pragmas and Syntax Trees

This release also included some language extensions which would allow users to write more efficient and readable code:

- support for *partially variadic functions* similar to those allowed in Python, allowing function expressions like `function( a, b, c, x... ) ... end;` which requires at least three arguments `a`, `b` and `c` and assigns the first three to `a`, `b` and `c` and then a list containing any remaining ones to `x`.
- more flexible list indexing, allowing users to install methods for multiple indexing (`m[i,j]` instead of `m[i][j]` for arrays) and more varied indices (such as `l[-1]` to access a doubly-infinite virtual list `l`). As well as being more flexible, this is a step towards supporting more efficient array implementations in which "row objects" (like `m[i]` in `m[i][j]`) might not exist.

GAP 4.8 had seven public releases between Month 6 and Month 24, and was replaced by GAP 4.9 in Month 33.

3.4. **Developments in GAP 4.9 (Month 33)**

GAP 4.9 continued the restructuring and modernizing of the core GAP system, aiming at the greater reliability, flexibility and performance which would be needed for GAP users to gain the full advantage of the OpenDreamKit ecosystem. For example, we removed our old home-grown large integer arithmetic implementation, and committed fully to the state of the art external library GMP. This move continues our cautious programme of exploiting existing free software in our core system. Our caution here is based on experience. Moving too quickly in

this area in the past has created problems with the reliability and portability of GAP on which our users depend.

This release incorporated a completely reimplemented build system for GAP, using more modern tools and complying with modern expectations. This resolved many issues with the old system, and made it easier to maintain and extend. For example it makes it much easier to maintain multiple configurations of GAP on a single system from a single copy of the source. Technical details of the new build system are described in `README.buildsys.md` from the GAP source code. Crucially the new build system was the final prerequisite for the planned merge of the HPC-GAP codebase into the mainstream GAP system as a compile-time option (for details, see Subsection 2.1).

An additional benefit of the new build system is that it makes it much easier to robustly include C or C++ extensions to the GAP kernel in contributed packages, including interfaces to many free C and C++ libraries. By making it easy to accelerate bottleneck steps, this greatly enhanced the overall performance of many packages, while encouraging integration of existing C libraries in packages complements our cautious approach mentioned above to including them in the core system.

One such example is the SEMIGROUPS package [23] by James D. Mitchell (USTAN). This started as a pure GAP package not requiring compilation, but a number of key algorithms from it were later converted into C and C++ in order to achieve the high performance necessary for its use in mathematical research. Rather than keeping these algorithms in a simple GAP kernel extension, they were made into a separate C++ library called LIBSEMIGROUPS [24], which does not rely on GAP or any of its packages. This library is a required component of the SEMIGROUPS package, and provides high-performance code to compute such objects as presentations, Cayley graphs and rewriting systems for finite semigroups, as well as information about semigroup congruences. Many parts of the library support multi-threading, meaning that users with multi-core computers can take advantage of parallelism even when not running HPC-GAP. Furthermore, the library can be accessed from Python or SAGE via a special set of Python bindings without any need for GAP to be loaded. The development of these bindings, other work on LIBSEMIGROUPS, and the mathematical research that came from it, are good examples of OpenDreamKit collaboration between USTAN and UPSud. An example of LIBSEMIGROUPS being called and used through GAP is shown in Appendix D.

Another example is the CURLINTERFACE package [18] by Christopher Jefferson and Michael Torpey, which first appeared in GAP 4.9.3 distribution. This package allows a user to interact with http and https servers on the Internet, using the CURL library, and is used by the PACKAGEMANAGER package [33] described in Subsection 5.6.

The GAP 4.9.1 release also incorporated the new ZEROMQINTERFACE package [29] by Markus Pfeiffer and Reimer Behrends, which provides both low-level bindings as well as some higher level interfaces for the ZEROMQ message passing library for GAP and HPC-GAP, enabling lightweight distributed computation.

The release of this package at this time allowed us to include in GAP 4.9.2 a further three packages. Firstly we have the JUPYTERKERNEL package [30] by Markus Pfeiffer, developed under WP4 and reported in deliverable D4.7, providing a so-called kernel for the Jupyter interactive document system. This package requires GAP packages IO, ZEROMQINTERFACE, JSON, and also two other new packages by Markus Pfeiffer called CRYPTING and UUID [26, 27], all included in the GAP 4.9.2 distribution.

GAP 4.9.3 also included the DATASTRUCTURES package [32] by Markus Pfeiffer, Max Horn, Christopher Jefferson and Steve Linton, which aims at providing standard datastructures, consolidating existing code and improving on it, in particular in view of HPC-GAP.

The GAP 4.9.1 release also incorporated many smaller performance improvements. For example, GAP now supports named constants, whose value cannot change at runtime. While

maintaining readability through naming, uses of these constants can be optimized as a function is parsed, so that they are at least as fast as using explicit values. Another notable improvement was to the widely used sorting functions which now use the best recently published algorithms and were also made easier to maintain by restructuring the implementation.

Debugging and profiling tools were also improved based on experience with GAP 4.8. Further changes improved GAP usability by allowing more efficient and/or readable code, in line with other modern programming languages:

- a concise syntax for short functions with any number of arguments, so that, for example, `{a,b} -> a+b` abbreviates `function(a,b) return a+b; end`.
- more flexibility in GAP expression syntax, where previously there were unintuitive technical restrictions. For example, an entry of the sum of two vectors can now be accessed as `x := (v+w)[i]` instead of `u := v+w; x := u[i]`.

GAP 4.9 had three public releases between Months 33 and 37, and was replaced by GAP 4.10 in Month 39.

3.5. **Developments in GAP 4.10 (Month 39)**

The release of GAP 4.9 with the new build system allowed the restructuring of the core GAP system to continue, and allowed us to finalize a number of other ongoing changes, leading to GAP 4.10 after a shorter-than-usual interval. GAP 4.10 is the current version of GAP, with the latest release being GAP 4.10.2, published in Month 46, and GAP 4.11 expected in November 2019.

The most important new feature in GAP 4.10 was the first public release of LIBGAP: an option to compile GAP as a C library instead of a stand-alone executable. This is very important for many uses of GAP in OpenDreamKit and was discussed in further detail in Section 3.1 below.

GAP 4.10 also provides experimental support for using alternative memory management systems, notably the Julia garbage collector and the Boehm garbage collector, in place of the GASMAN system which was developed for GAP 4 in the 1990s. These garbage collectors are thread-safe (important for HPC-GAP) and the option to use the Julia garbage collector will allow tight integration of GAP with Julia programs and other systems via Julia. Use of Julia in this way is an initiative led by the OSCAR project, developed within the DFG-funded grant SFB-TRR 195 “Symbolic Tools in Mathematics and their Application” and is seen as complementary to OpenDreamKit.

The work on improving debugging and profiling in GAP has continued. Profiling now reports memory usage as well as CPU time, and additional checking options (both internal and by using external tools such as VALGRIND) help to detect misuse of memory in the kernel.

Using the improved debugging and profiling facilities we were able to detect many performance bottlenecks and confirm that new code genuinely improved performance, as well as detecting many bugs much earlier in the development process. To alleviate performance bottlenecks in key applications, we rewrote much of the method selection (dispatch) code from GAP to C and gained large speedups by re-engineering many integer arithmetic functions.

For example, the table below gives the CPU time in milliseconds required to run the following calculation:

```
for i in [1..k] do x:= 1/p^q mod n; od;
```

p	q	n	k	ms, before	ms, after
3	60	2^{59}	2000000	2364	1219
3	60	2^{60}	2000000	2821	1292
2	6000	3^{6000}	6000	19358	692

The same improved profiling tools identified other "hidden" bottlenecks in GAP code. For instance GAP has a sophisticated system of inference that can trigger so-called "immediate methods": quick attempts to deduce additional consequnces from information that becomes known about an object. In general, these are very effective, and the deduced information can lead the system to chose better methods for subsequent calculations. For instance, when a group learns its order, if that order is finite and odd, then by the celebrated Feit–Thomson theorem the group must be solvable. Once that is known, much more efficient methods can be applied for many subsequent calculations.

While this is a powerful feature in most situations, in some calculations, a large number of short-lived objects are created, and the immediate methods use a great deal of time deducing knowledge about each of them that will never be used. The profiling developments made this easily visible for the first time, and some small adjustments to the immediate methods produced up to a six-fold speed-up in fundamental operations such as computing isomorphism groups. For instance, the run-time for the code fragment below dropped from 130 seconds to 22.

```
G:=PcGroupCode(
  741231213963541373679312045151639276850536621925972119311,
  11664);;
IsomorphismGroups(G,PcGroupCode(CodePcGroup(G),Size(G)))<>fail;
```

We have also continued to develop package JUPYTERKERNEL following its inclusion in GAP 4.9 to enhance its usability. The advent of Binder allowed us to give users an opportunity to try GAP Jupyter notebooks online and share reproducible GAP experiments (see Appendix C).

The inclusion of the JUPYTERKERNEL package and its associated dependencies into GAP 4.9 facilitated the appearance of new packages extending its graphical capabilities. GAP 4.10.0 incorporated the FRANCY package [22] by Manuel Martins and the JUPYTERVIZ package [7] by Nathan Carter[2] which allow users to create and work with charts and graphs. The latter package also allows the user to produce graphics from the GAP command line.

3.6. **Developments in GAP 4.11 (anticipated in November 2019)**

At the time of preparation of this report, the next major release, GAP 4.11, is approaching completion. Almost all of the features to be included in it (including all the ones highlighted here) are already publicly available on GitHub to developers or anyone else interested.

One important and relevant feature of this release will be the "syntaxtree" reflection API for GAP, allowing GAP programs to access and modify the internals of GAP functions and opening up a range of possibilities for optimization, automatic parallelization, and more effective compilation into C. This is discussed in more detail in Section 3.2 above.

Additional relevant features are continued improvements to debugging and profiling facilities (for example, adding support for profiling interpreted code and making profiling reports more informative by giving more library methods human-readable names; and enhancements to the testing infrastructure designed to make the test suite more robust and easier to run, and tests more independent and easier to write).

The LIBGAP API has been extended to allow programs using the system in this way more convenient access to GAP objects such as characters, floats, integers and matrices; the performance of Julia GC integration also improved, and the memory usage on Windows when running external programs was reduced.

[2]Nathan Carter visited USTAN on sabbatical from Bentley University in 2018–2019, and was employed part time to work mainly on WP3. In addition to JUPYTERVIZ, he also carried out a major redesign of his GROUPEXPLORER software – a popular teaching resource, accompanied by the textbook [10], which had been developed in the previous decade and was in an urgent need of modernization. The new GROUPEXPLORER version, available at [9], also allows a user to load and run the GAP code to construct and explore groups from the GROUPEXPLORER library interactively, launching a GAP session on a SAGEMATH cell server.

Further performance enhancements included, among others, speeding up writing to global variables; optimizing operations involving identity permutations; speeding up `IsConjugate` for `IsNaturalSymmetricGroup`, improving performance of `NormalizerViaRadical`, and of `ConjugacyClasses` for solvable groups. For example, the time of the following calculation reduces from 110 seconds to 53 seconds:

```
G:=WreathProduct(CyclicGroup(IsPermGroup,12),
       DihedralGroup(IsPermGroup,12));
ConjugacyGlasses(G);
```

As one example of the general effectiveness of our performance improvements, the full test suite of GAP 4.7.9 (current at the start of the project) which takes 3237 seconds to run on GAP 4.7.9 on current hardware, runs unchanged in just over 2900 seconds with a pre-release version of GAP 4.11. Many of the most widely used and important areas of GAP functionality have seen bigger changes.

4. Additional Developments in GAP packages

Other sections of this report describe a number of GAP packages such as MEATAXE64 and PROFILING that were developed, published, and/or added to the GAP distribution over the duration of the project. In this section we briefly list some other notable new or improved packages that contribute to our overall goals of making GAP more efficient, usable, composable and scalable, and enabling users to solve larger and more complex problems.

- DIGRAPHS by Jan De Beule, Julius Jonušas, James Mitchell, Michael Torpey and Wilf Wilson [12] provides very efficient C-based functionality for fundamental computations with directed and undirected graphs, including those with multiple edges (added in GAP 4.8.2 and repeatedly extended). These algorithms underpin many mathematical computations, especially in semigroups and combinatorics.
- FERRET by Christopher Jefferson [16] provides a C++ reimplementation of Jeffrey Leon's Partition Backtrack framework for solving graph-isomorphism like problems in permutation groups (published as a package submitted for the redistribution with GAP). This has the effect of accelerating many standard group-theoretic computations.
- FINING by John Bamberg, Anton Betten, Philippe Cara, Jan De Beule, Michel Lavrauw and Max Neunhöffer [3] for computation in Finite Incidence Geometry (added in GAP 4.8.2). Computations in this area involve extremely large combinatorial searches, making them an interesting application for parallel computing.
- MAJORANAALGEBRAS by Markus Pfeiffer and Madeleine Whybrow [31] constructs Majorana representations of finite groups (added in GAP 4.10.1). This is interesting primarily as an application of high performance kernels. This whole area of mathematics was inspired by the Griess algebra used to construct the celebrated "Monster" sporadic simple group, which has dimension 196884. Many of the interesting problems in this area have similarly high dimension and HPC-GAP and MEATAXE64 methods have both been used to good effect in this area.
- MATGRP by Alexander Hulpke [15] provides an interface to the solvable radical functionality for matrix groups, building on constructive recognition (added in GAP 4.8.2). This improves the general capabilities of GAP in this area, and reduces many key problems to linear algebra and/or partition backtrack questions, which can exploit the power of MEATAXE64 and/or FERRET.
- NORMALIZINTERFACE by Sebastian Gutsche, Max Horn and Christof Söger [14] provides a GAP interface to Normaliz, enabling direct access to the complete functionality of Normaliz, such as efficient computations in affine monoids, vector configurations, lattice polytopes, and rational cones (added in GAP 4.8.2).

- SCSCP by Alexander Konovalov and Steve Linton [19] migrated to GitHub and had several releases in Months 18–45, which included improved testing of master–worker communication due to the use of CI tools and new functionality required in WP6 as well as code refactoring and bugfixes. Its usability for coarse-grained distributed memory parallel computation (see `https://github.com/alex-konovalov/scscp-demo` for a short tutorial) makes it a suitable replacement for the ParGAP package by Gene Cooperman, which has not been redistributed with GAP since GAP 4.9.
- SemigroupViz by Nathan Carter [8] is an emerging package which is based on the JupyterViz package [7] and provides tools to visualise egg-box diagrams and Cayley graphs in GAP. It can be used in Jupyter notebooks or from the GAP command line.
- Walrus by Markus Pfeiffer [28] provides methods for proving hyperbolicity of finitely presented groups in polynomial time (added in GAP 4.10.1). This again is an application area for High Performance methods in GAP, requiring some very unusual data structures and search primitives.
- YangBaxter by Leandro Vendramin and Alexander Konovalov [34] provides functionality to construct classical and skew braces, and also includes a database of classical and skew braces of small orders (added in GAP 4.10.1). Braces are algebraic structures related to the solutions of the set-theoretic version of the celebrated Yang–Baxter equation, initially introduced in statistical mechanics. This database is a powerful tool for testing conjectures in this actively developing area. Extending it for larger orders would involve extensive computations, for which we plan to use parallel computations with SCSCP and HPC-GAP.

5. Developments in GAP infrastructure: Tools and Processes for System and Package Development, Quality Assurance and Release

As well as developments in the core system and in packages, we have dedicated considerable effort to improving our infrastructure and processes. In this section we describe developments in the technical processes which support our developers (especially package authors and maintainers). These processes connect developers to our user community and aim to ensure the robustness and quality of the integrated system, something essential to its use in demanding large-scale computations. Some of the technical developments (especially around continuous integration) were reported in D3.8 (Month 36), and some of the associated advocacy, training and support in deliverables D2.2, D2.11 and D2.15. This section updates and extends those focused reports, taking a broad view of our tools and processes and focusing on the visible benefits of these changes to developers and package authors. In subsection 5.5 we assess the impact that this work has had on the developers and package authors.

5.1. **Regression Testing**

Regression testing checks (preferably in an automated way) that changes and additions to a system do not break functionality that worked previously. If a change breaks a test, that is a *regression*. In GAP, regressions may occur when a change causes an incorrect result, or a crash, or an unwanted error message; in addition, there may be *performance regressions* and *memory regressions* where a test becomes slower, or uses much more memory after a change.

Regression testing has been used by GAP in one or another form since 1980s, and by the beginning of the OpenDreamKit project, GAP had established its testing infrastructure based on a private Jenkins installation at USTAN, not accessible outside the local network. That was a major limitation, because sharing test outcomes with other developers and allowing them to repeat tests in the same environment was cumbersome and time-consuming with many manual steps.

During OpenDreamKit there has been widespread adoption of open development models by the GAP system and package developers and maintainers. As a result, we have also been able to move to an open regression testing infrastructure, which has enabled us to substantially improve GAP development and testing workflows and develop a more robust system with less effort. For example, today, anyone who is interested in checking the status of the GAP test suite can visit the public dashboard at `https://github.com/gap-system/gap-distribution/` of which a snapshot is shown in figure 7. This dashboard shows the status of the core system GAP tests, which are run for every change made to the repository, as well as package integration tests described below.

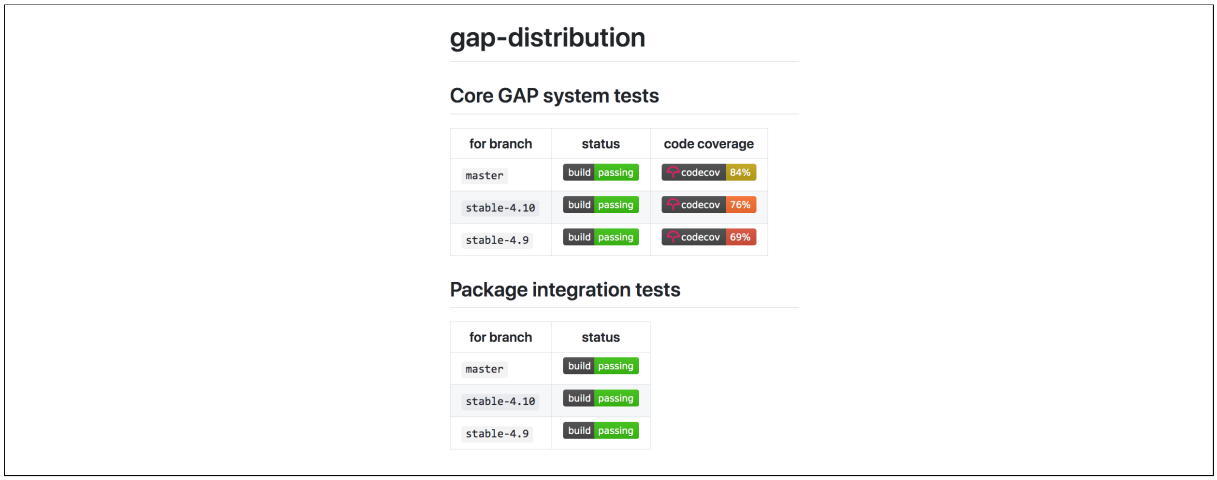


FIGURE 7. Dashboard with core GAP system and package integration tests

The "status" lights are actually active buttons, which lead to the full test reports, held on the public continuous integration platform Travis CI. The "code coverage" lights are also buttons and lead to the public Codecov service, where, if you drill down far enough, you can access reports showing exactly which lines of code in the kernel and library have, or have not been tested.

This level of open continuous testing has led to huge improvements in the quality of the system, with far fewer bug reports than previously, and saved enormous effort in integrating releases.

At the heart of the testing of the core GAP system is our comprehensive test suite. The table in figure 8 shows its increase over the period. In addition, we also test the correctness of the over $14,000$ lines in the 1810 examples in our manuals. We promote a similar testing approach to GAP package authors and encourage them to provide their own regression tests. Many do, adding, in total over a quarter of a million lines of further tests in the distributed system, which are run when new versions of packages and/or of the core system are being tested for integration into a release.

GAP release	Release date	Number of test files	Number of lines in test files	Code coverage
GAP 4.7.8	November 2015 (Month 3)	69	21981	N/A
GAP 4.8.10	January 2018 (Month 29)	91	28586	N/A
GAP 4.9.3	September 2018 (Month 37)	564	41456	69 %
GAP 4.10.2	June 2019 (Month 46)	648	53362	76 %
GAP 4.11	November 2019 (planned)	707	61280	84 %

FIGURE 8. Growth of the GAP test suite

5.2. **Docker Containers for Testing, Using and Sharing GAP Code**

We have continued to maintain and expand the range of Docker containers, initially reported in D3.1 and then in D3.8. Our main applications of these containers are:

- speeding up regression tests on Travis CI;
- sharing reproducible experiments in GAP Jupyter notebooks running on Binder (see Appendices C and D);
- providing an alternative distribution. This particularly helps users who may not have administrator access to their systems, or to access packages that they cannot run otherwise (e.g. due to missing dependencies or incompatibility with the operating system).

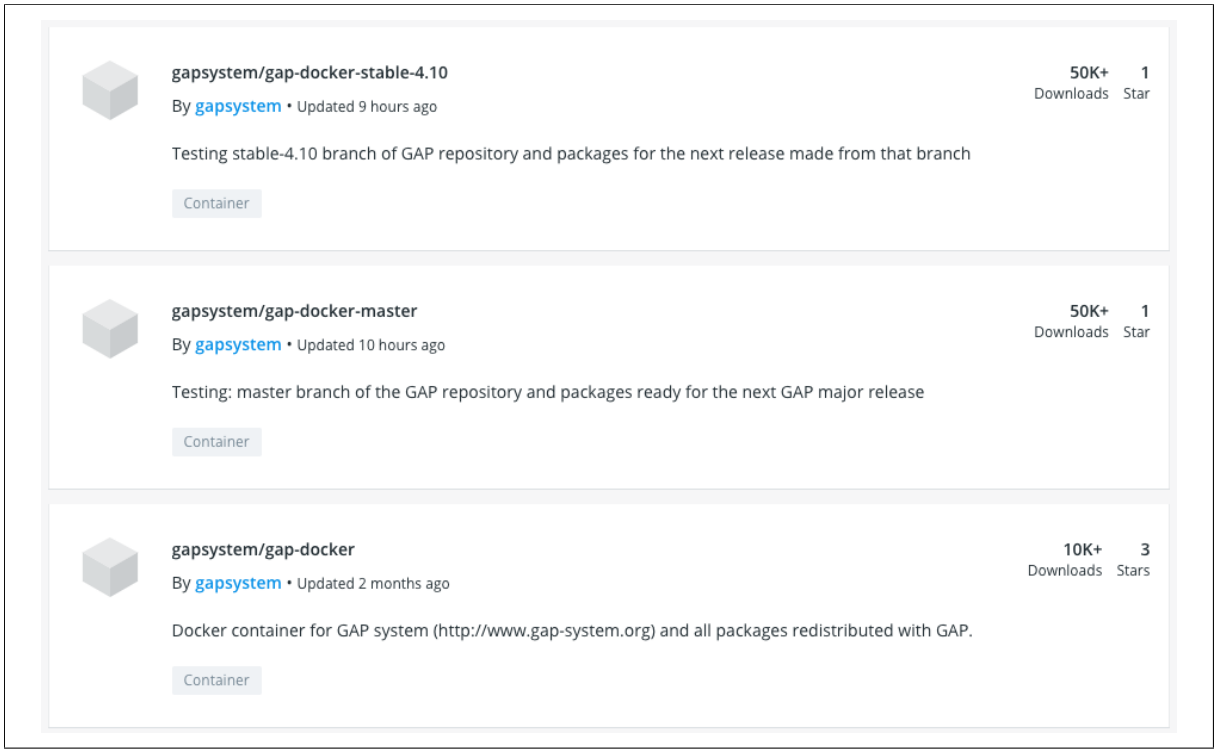


FIGURE 9. Selected GAP Docker containers on Docker Hub

These containers are publicly available on Docker Hub (see Figure 9). Containers with the development versions of GAP and packages are updated daily, and have the largest number of downloads since they are regularly used for testing. An example of a test using such container could be found under "Package integration tests" in the dashboard shown in Figure 7. Clicking on the "status" button leads to the overview displayed in Figure 10, from where one could inspect test output for each of the configurations.

5.3. **Continuous Testing of Package Cross Compatibility Ahead of GAP Releases**

For packages redistributed with GAP, our automatic package update system checks regularly for new versions on the authors web pages, retrieves them, and then uses them in a number of checks to ensure that new package releases are compatible with each other and do not break the functionality of the core GAP system. The same process also helps us to check that changes in the core GAP system do not break the functionality of the packages redistributed with GAP. This system has dramatically simplified the process of making a GAP release or update, for which we need a mutually compatible set of package versions, also compatible with the new core system. This used to require extensive negotiation with package authors, and sometimes took months, even though the number of packages was much smaller. Now developers have continuous visibility of the functionality and compatibility of released and upcoming versions of

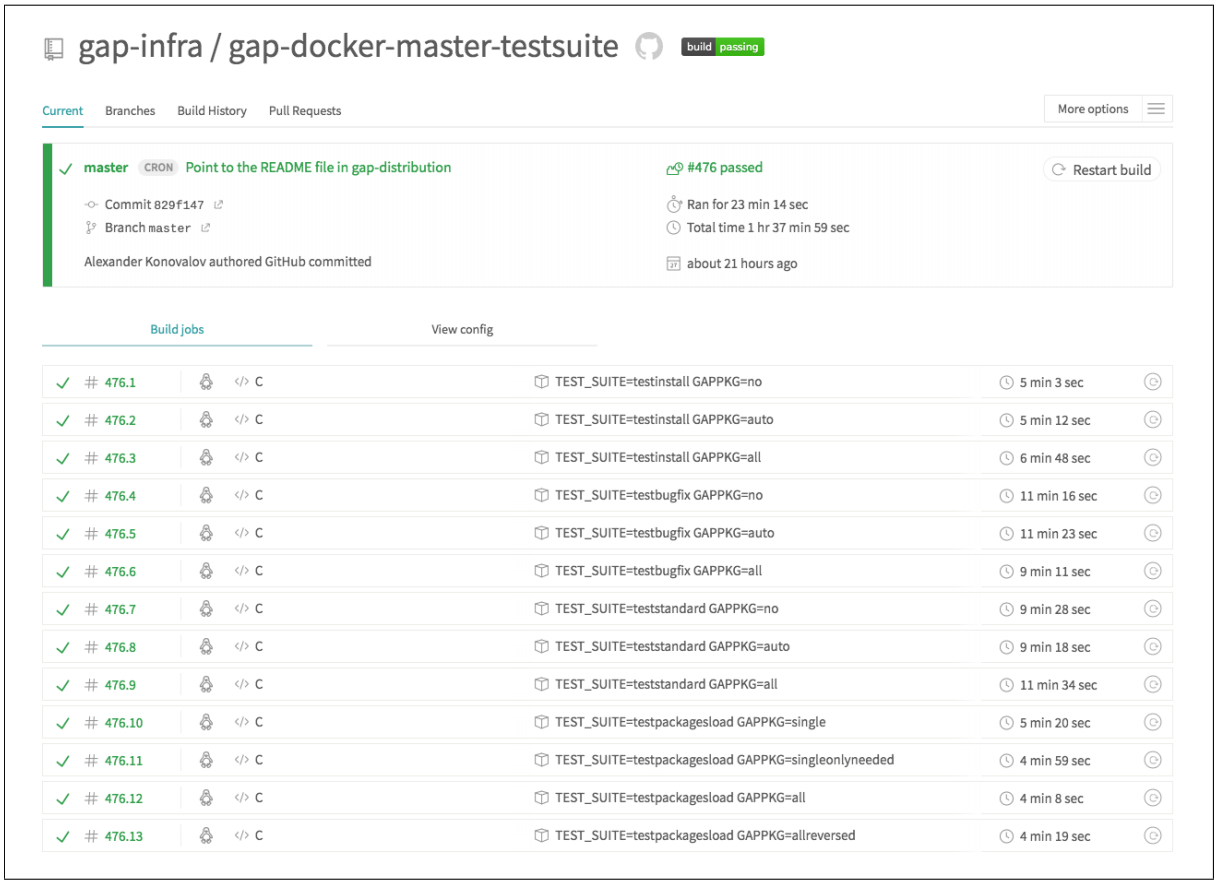


FIGURE 10. GAP package integration tests on Travis CI

packages and of the core system, and releases are much faster, delivering new features to end users without compromising speed of delivery or robustness.

5.4. **Improvements to GAP Package Tools**

Sections 4 and 2.2 among others have shown that developments important to our goal of a flexible, widely portable, high performance mathematical software system can increasingly occur in packages, as well as in the core GAP system. For that reason, as well as pursuit of the broader goals of OpenDreamKit to support a user/developer community, we have attached considerable priority to a supporting and promoting a healthy GAP package ecosystem. We have supported this through the developments described in subsection 5.1, together with community supporting activities described in D2.15, with good effect. During the OpenDreamKit project we have observed rapid growth of the number of GAP packages and of the number of active package authors as well as steady improvements in the quality of package testing and more frequent package updates. This has both been supported by, and encouraged, the development of a wide range of tools to help package authors, which we describe below.

5.4.1. *ReleaseTools and GitHubPagesForGAP.* Our collaborator Max Horn (University of Siegen) developed RELEASETOOLS[3] and GITHUBPAGESFORGAP[4] which allow package authors to fully automate the release procedure of a GAP package hosted on GitHub and the maintenance of a website for it hosted on GitHub pages, reducing the time needed to publish it to a matter of minutes.

[3] `https://github.com/gap-system/ReleaseTools/`
[4] `https://github.com/gap-system/GitHubPagesForGAP/`

5.4.2. *The Example Package.* The GAP package EXAMPLE by Werner Nickel, Greg Gamble and Alexander Konovalov [25], which acts as a package template, has been consistently maintained to track the development of the packages infrastructure and serves to demonstrate a model use of modern development tools.

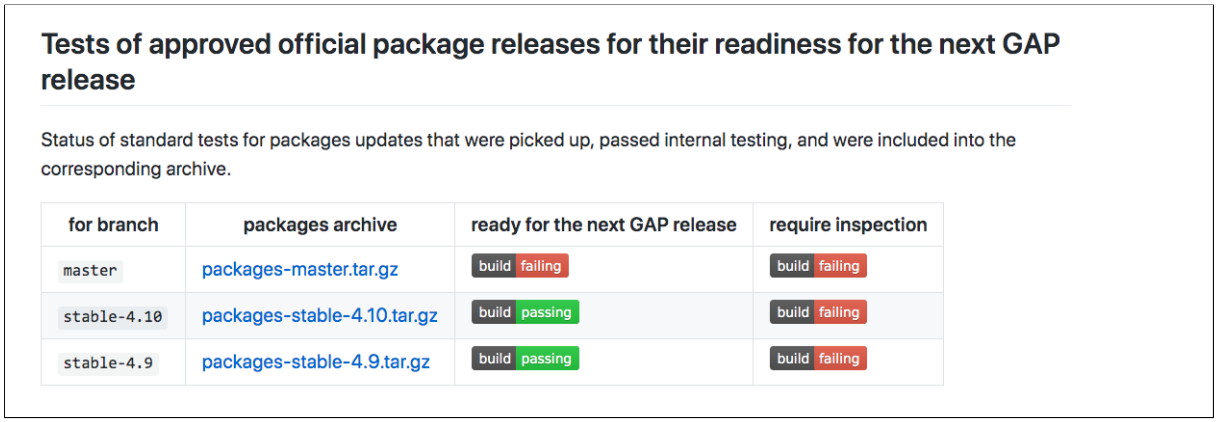
5.4.3. *PackageMaker.* To offer an alternative way to create a package, Max Horn developed a GAP package called PACKAGEMAKER[5] (not yet redistributed with GAP), which provides an interactive "package wizard". This allows a user to fill in the basic details of the intended package and then creates all of the files making up the basic structure of the package. A particularly relevant strength of this tool is that it hides the additional complexities of packages which include C or C++ code, removing a barrier to faster implementation of key kernels.

5.4.4. *AutoDoc.* GAP packages are required to provide documentation. The key central format for GAP documentation is XML-based, defined in the GAPDoc package, and while convenient in many ways, can be rather cumbersome and error prone to write by hand. In 2012 Sebastian Gutsche and Max Horn (both now at the University of Siegen) developed the AUTODOC package [13] which creates documentation from simple structured comments in the source code, similar to PYDOC or JAVADOC.

By now, AUTODOC has become very reliable and configurable, and is in use by 89 GAP packages.

5.4.5. *Continuous Integration Support for Package Authors.* Using the Docker containers described in subsection 5.2 and shown on Figure 9 enabled us to make the results of regression tests for GAP packages publicly available via Travis CI. Using a prebuilt Docker image greatly reduced the runtime of the test and allowed us to have a separate test configuration for each package, which is easily discoverable via the Travis CI interface.

Figure 11 shows a fragment of the dashboard from `https://github.com/gap-system/gap-distribution` which indicates tests status. When this snapshot was taken, it happened that one package was depending on undefined behaviour of a GAP library function, which was changed, resulting in package tests failing. The prompt detection of this interaction made it easy to diagnose and the author was immediately notified and reacted with a quick fix. Previously this could have required many weeks of tedious correspondence to resolve.



Tests of approved official package releases for their readiness for the next GAP release

Status of standard tests for packages updates that were picked up, passed internal testing, and were included into the corresponding archive.

for branch	packages archive	ready for the next GAP release	require inspection
`master`	packages-master.tar.gz	build failing	build failing
`stable-4.10`	packages-stable-4.10.tar.gz	build passing	build failing
`stable-4.9`	packages-stable-4.9.tar.gz	build passing	build failing

FIGURE 11. Dashboard for Travis tests of GAP packages (GitHub view)

One of the extensions of the package testing framework, greatly facilitated by packages migrating to GitHub, was adding tests not only for the latest official releases of GAP packages, but also for their *development* versions. In this case, the onus to check test outcomes is on package authors, but they are able to quickly see how the changes that they have made to a

[5] `https://github.com/gap-system/PackageMaker`

package, but not released yet, work in different settings (the released and development versions of GAP, with various different combinations of other packages, for instance). Again this leads to prompt diagnosis and easy fixing of problems, and improves the quality of released packages.

Finally, Figure 12 show the code coverage leaderboard. We aim at ensuring that code coverage for packages is not worse than for the core GAP system, but in fact many packages put a lot of effort in ensuring that their coverage is close to 100%.

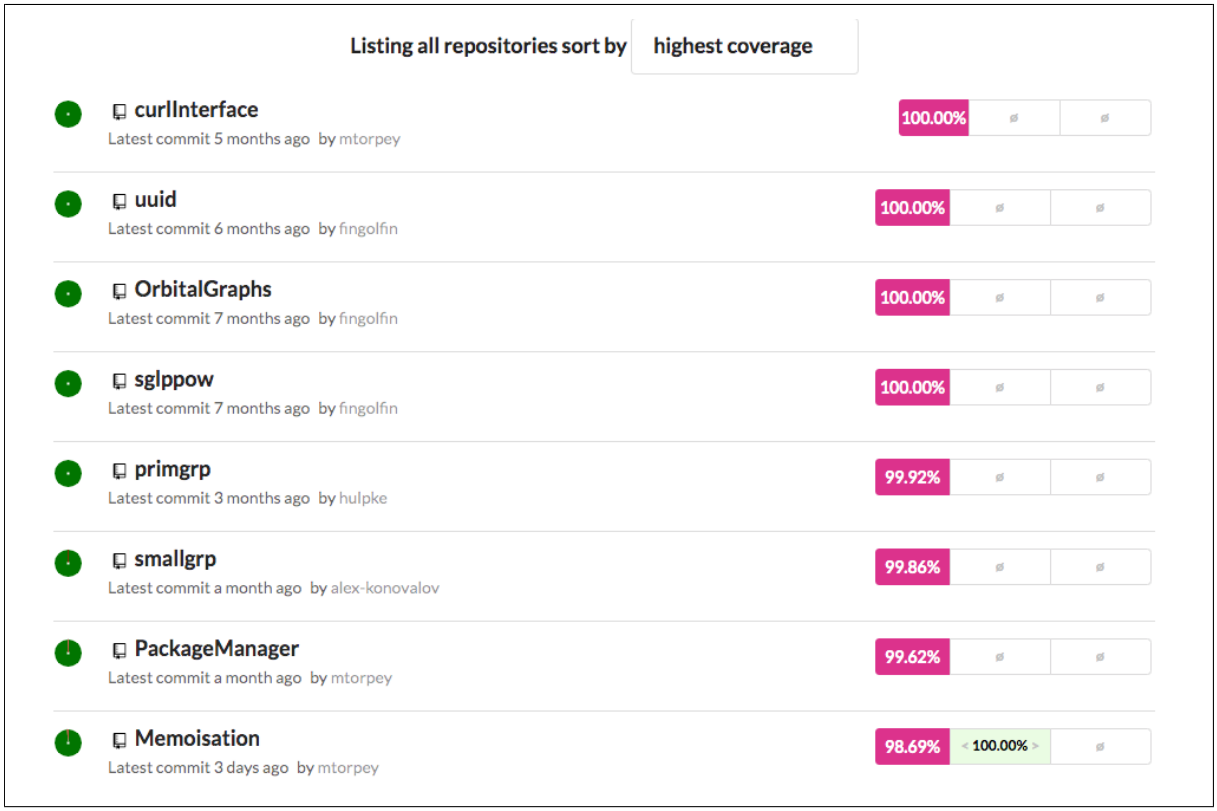


FIGURE 12. Codecov leaderboard for in gap-packages GitHub organization

5.5. **Open Development of GAP Packages – Measure of Uptake and Quality**

As we have explained above, we encourage and support package authors to use open development models and tools similar to those used for GAP itself, since we believe that this is the best way to offer our users a large portfolio of high quality compatible packages which meet their diverse computational requirements.

The openness of our preferred model also provides a number of tools that we can use to measure our success, both in encouraging uptake and in some aspects of the quality of the packages.

At the time of writing only 4 packages redistributed with GAP do not have a public source code repository (to compare, in Month 38 there were still 23 such packages). In some cases, reaching this point required significant efforts on our part. We had to establish contacts with authors who left academia, gain their permission and agree a suitable license. We could then populate a repository with a history based on archives of past releases. In these cases (where we deem the package important enough, we formally "adopt" the package and declare it to be collectively maintained by the GAP Group. This work helps ensure that the code will be preserved in a usable state as GAP and other tools evolve, which is necessary to check and reproduce computational results. It also gives visibility to the packages, sometimes attracting new contributors or even maintainers. As Figure 14 shows, the number of individuals listed as authors of GAP packages increased from 158 in Month 3 to 196 in Month 46.

Figure 13 shows the number of GAP packages included in some GAP release by year (of the release). At the beginning of the project, GAP 4.7.9 published in November 2015 included 119 packages, 52% of which had published a release in 2014–2015. Now, GAP 4.10.2 includes 145 packages, 88% of which have published a release in 2018–2019. The "long tail" of packages that haven't been updated for a long time is gradually reducing, a very encouraging sign of the vigour of the community.

The larger proportion of packages with recent releases shows us that more packages are being updated to make use of the recent GAP developments, and that package maintainers are responsive to reports about detected problems. This is greatly facilitated by hosting packages on GitHub, making it easy and convenient for GAP developers to offer support in the form of code patches ("pull requests" in GitHub terminology) or even by actually publishing releases of packages.

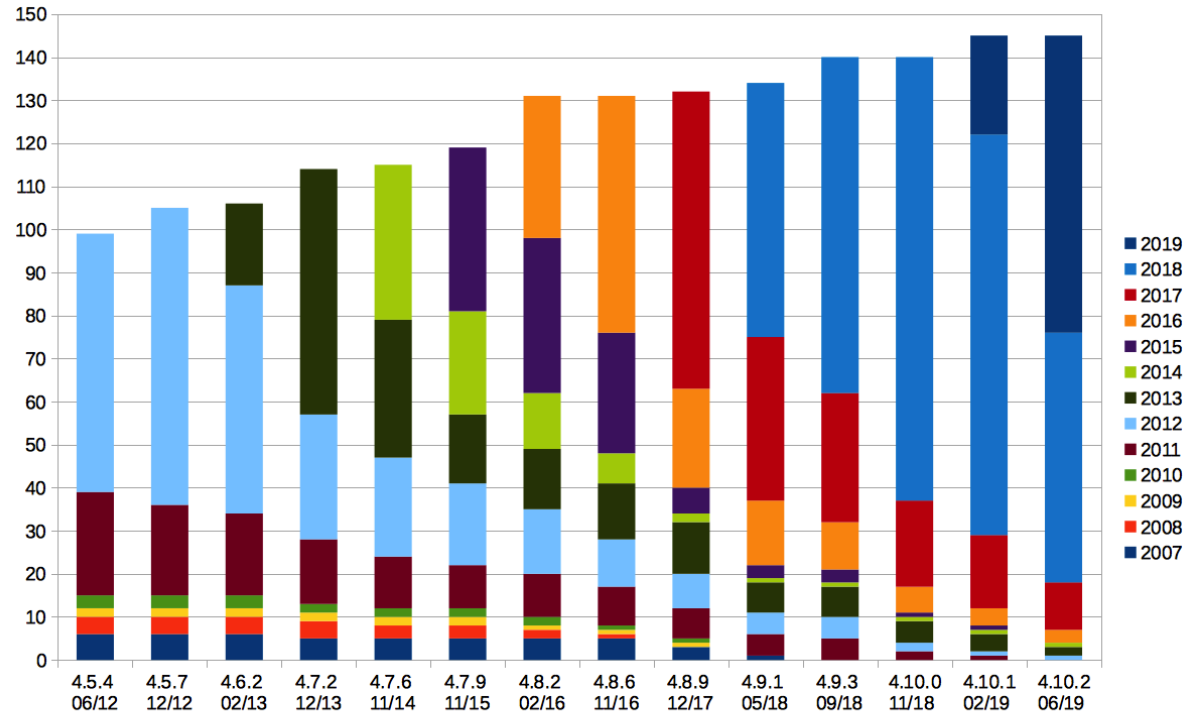


FIGURE 13. Number of GAP packages and their release year

Over the same period, package releases not only became more frequent, but their quality also improved. In November 2015, 52% of packages did not provide a standard test suite at all, whereas now 78% of packages have one. It is then easy for their authors to regularly test them with published GAP releases and with the development version. Some have their own continuous integration systems (e.g. the homalg project, which uses CircleCI). At the moment of writing, in the stable-4.10 branch, standard tests pass for 102 packages and fail for 12. These failures are typically not serious bugs but variations in output format, or an incompatibility with some other package that is not loaded by default.

Another illustrative statistics is the number of GAP packages whose websites are hosted on GitHub pages. This is not a specific suggestion or requirement, but it indicates engagement with open development methods, and makes it easy to keep the website up to date. The following table gives this number for August of each respective year:

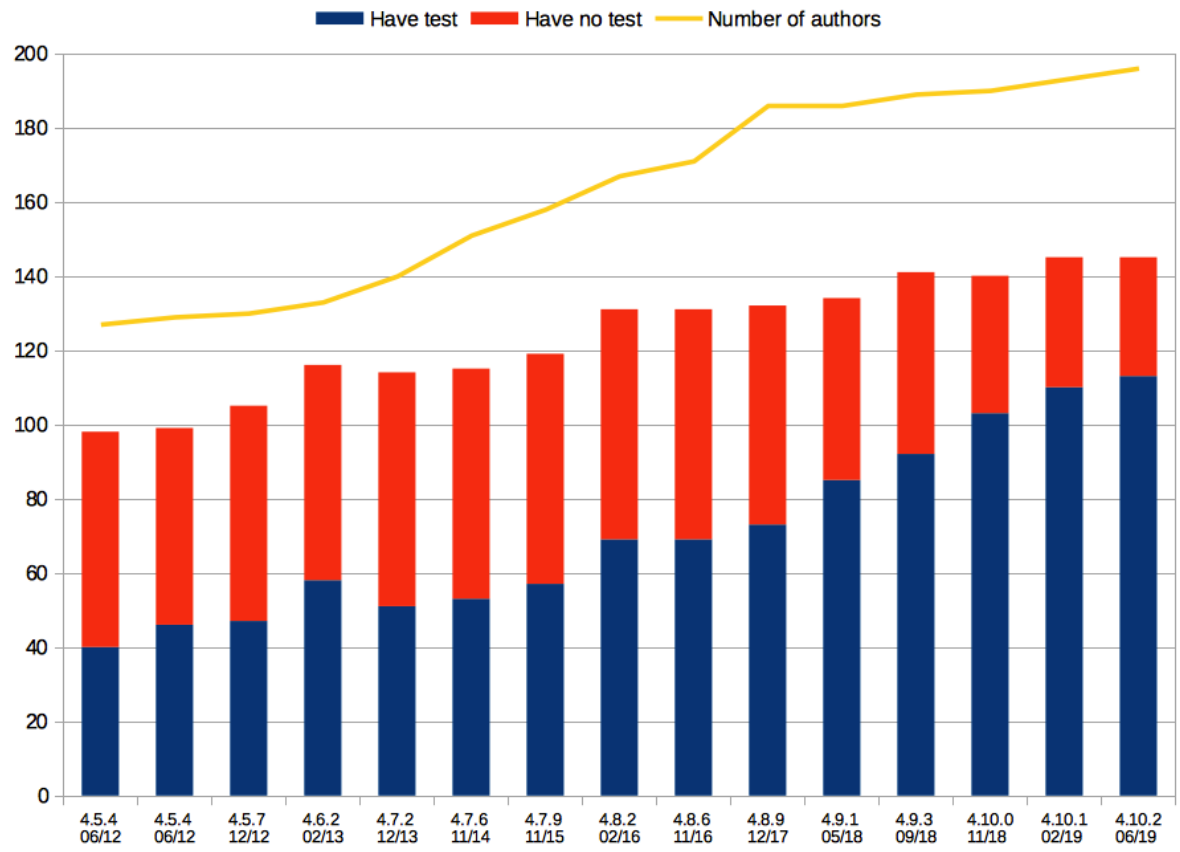


FIGURE 14. Number of GAP packages, their authors, and packages with tests.

Year	2015	2016	2017	2018	2019
Number of packages	14	45	61	92	124

5.6. **The GAP Package Manager**

5.6.1. *Background.* For many years, the installation of packages in GAP has been an entirely manual process. If a user requires a package that is not currently installed on their system, they are required to visit the GAP website, or the website of the required package, download an archive or repository and extract it into their package directory. In many cases, they must also compile C code or external programs within the package before use, in a process that is not consistent between packages. Worst of all, any packages on which their chosen package depends must also be installed individually, including all these steps for each one.

This awkward installation procedure is mitigated by the inclusion of a large number of packages with the main release archive, which solution masks the problem for the majority of users, who do not require any more specialized packages. However, it also results in a 1.6 GB standard installation, of which the packages make up 1.4 GB, and while the files are installed, C code often does not get compiled. While some packages are used by almost every GAP user because they enhance the performance of core GAP features, only a subset of the others will be needed by any specific end user. This need may be explicit (when they load a package relevant to their research interests) or implicit (to satisfy package dependencies). It would be desirable to avoid all users having copies of all packages by default.

Additional problems can arise when the GAP installation is controlled by central administration and a user needs an extra package installed or a package compiled, or when a user wishes to update to a new package version ahead of it being included in a new GAP distribution.

In the light of these issues, the possibility of a more sophisticated package management system for GAP has been discussed for some time, and was realized with the creation of PACKAGEMANAGER [33] in Month 37, by Michael Torpey. Development started after a discussion at GAP Days Fall 2018 (`https://www.gapdays.de/gapdays2018-fall/`) and an initial release was made before the end of that meeting. New features have been added in various further releases across the past year.

5.6.2. *Design.* PACKAGEMANAGER is a GAP package itself, and is utilized in the standard way, by loading it into a GAP session and executing GAP functions. This approach to package management was chosen over an external manager since it allows users to install and remove packages without ever leaving the GAP prompt, while making no, or minimal assumptions about the underlying system setup. This gives users a comfortable and familiar environment. In most cases, packages can be installed and loaded without restarting the session, as shown in Figure 15. This means that users' workflow is interrupted as little as possible, and they do not need to save and reload their work.

```
gap> InstallPackage("digraphs");
#I  Getting PackageInfo URLs...
#I  Retrieving PackageInfo.g from https://gap-packages.github.io/Digraphs/PackageInfo.g ...
#I  Downloading archive from URL https://github.com/gap-packages/Digraphs/releases/download\
/v0.15.4/digraphs-0.15.4.tar.gz ...
#I  Saved archive to /tmp/tmcLM2FI/digraphs-0.15.4.tar.gz
#I  Extracting to /home/user/.gap/pkg/digraphs-0.15.4 ...
#I  Checking dependencies for Digraphs...
#I    io >=4.5.1: true
#I    orb >=4.8.2: true
#I  Running compilation script on /home/user/.gap/pkg/digraphs-0.15.4 ...
true
gap> LoadPackage("digraphs", false);
true
```

FIGURE 15. Installing a package using PACKAGEMANAGER.

The most important function provided by PACKAGEMANAGER is `InstallPackage`, which can take as argument the URL of an archive, repository, or package info file, or simply the name of a package. In the case of a URL, the appropriate file is retrieved using an Internet connection, and used to install the package; if a package name is given, it is looked up in a pre-defined list of package URLs and installed from there. In most cases, the latest released version of a package is installed, but a user can specify an older version by giving an explicit archive URL, or a development version by specifying a branch in a version control repository. Also provided are `RemovePackage` and `UpdatePackage`, which remove a package or update a package to the latest version, given the package's name.

The ability to install a package given only its name is useful, but requires some important design decisions: how do we decide which packages to support, and which versions do we install by default? One option would be to have a centralized repository (as is common for the APT package manager) which contains a carefully checked set of package versions that are known to work with the installed version of GAP and with each other; this has the advantage of stability and control, but may make it harder to get the latest software versions. The other option is to store links to appropriate locations on the package websites, allowing the most recently released version to be installed without explicit approval from the GAP authors; this approach is closer to the more liberal PIP package manager for the Python language. In the end, the latter approach was taken for the following reasons:

- approved package versions for a given release are likely to be included with the GAP installation anyway, and it is unlikely that users would need to reinstall these;

- the latter approach requires no additional infrastructure, since a list of URLs for package info files is already maintained for testing purposes;
- this approach encourages active use, and therefore testing, of recent releases.

This decentralized approach has therefore been adopted for the moment, though future development could move easily towards a different approach.

5.6.3. *Features.* Over the course of PACKAGEMANAGER's development, it has accumulated various features that automate as much of the installation process as possible. During installation, the following operations are carried out automatically:

- any kernel module is configured and compiled, if present;
- package documentation is built from source if necessary;
- external software prerequisites are installed using a package-specific shell script, if one is present;
- all missing package dependencies are installed, or recompiled if broken;
- basic checks are performed to verify that installation was successful.

The new packages are installed in a user-specific directory separate from the GAP installation, so that multiple users on a shared system can have distinct package configurations. The packages will also remain present if the main GAP installation is moved or replaced.

This has resulted in a reasonably smooth installation process, with very little friction encountered by a user who tries to get a new package working. The package manager is now a deposited package in GAP, and is therefore distributed in the main archive with each release.

5.6.4. *Best Practice.* PACKAGEMANAGER was written using modern open-source software technologies and practices. In the report on OpenDreamKit D1.5, in Section 4.3.2, there are two checklists for best practices: one for software engineering, and one for dissemination. The package fulfills all the requirements in both lists, as summarized in Tables 16 and 17.

Version control	✓	Git
Tests	✓	GAP test suite with 99% code coverage
Automated tests	✓	
Continuous integration	✓	Travis runs test suite for every commit
Automatic building of releases	✓	RELEASETOOLS (see above)

TABLE 16. Software engineering good practice checklist for PACKAGEMANAGER

5.6.5. *Other Uses.* Aside from making the on-demand installation of individual packages easier, PackageManager has implications for the future of how GAP is distributed. There are no current plans to stop distributing deposited packages with the GAP archive by default, but the possibility of distributing a much smaller archive, containing only a few fundamental packages and PackageManager itself, and then installing others as needed, now opens up. Furthermore, the package manager can be used by other systems that interact with GAP to manage their GAP installations more easily. The COCALC system, for instance, now contains PACKAGEMANAGER as part of its GAP 4.10.2 installation, enabling each user to install any packages they need without COCALC operators being forced to manage these installations separately. This is an example of positive two-way interaction with COCALC, and sets as a good precedent for interaction with other systems.

Host code publicly	✓	`https://github.com/gap-packages/PackageManager`
Reference Manual (APIs)	✓	GAPDoc/Autodoc, and hosted on package website
Tutorial (for beginning users)	✓	Quick examples in readme and manual, before more detailed documentation
Examples	✓	
Live interactive online demos	✓	Binder link included in readme
Support mechanisms	✓	Github issues
How to cite the output?	✓	Explained on readme and website
Installation mechanism	✓	Distributed with GAP, and can update itself
High level description accessible to non-experts	✓	In readme, and chapter 1 of manual
URLs/Blog/etc to and from OpenDreamKit project	✓	OpenDreamKit link and logo in readme
Grant acknowledgments	✓	Acknowledged in readme
Open Source license	✓	GPL v2 or later
Workshop	✓	Demos in *CIRCA Seminar* (USTAN, Month 43) and *Workshop on Mathematical Data* (Cernay, Month 48)
Engaging users	✓	

TABLE 17. Dissemination good practice checklist for PACKAGEMANAGER

APPENDIX A. HPC-GAP– REFERENCE MANUAL

This Appendix contains a copy of one of the four main GAP manuals from the release of GAP 4.10.2 (Month 46).

This manual contains documentation for the multi-threaded programming in GAP, described in Section 2.

A current version of GAP Documentation in PDF and HTML formats could be found on the GAP website at `https://www.gap-system.org/Doc/manuals.html`.

HPC-GAP — Reference Manual

Release 4.10.2, 19-Jun-2019

The GAP Group
Reimer Behrends
Vladimir Janjic

The GAP Group Email: support@gap-system.org
Homepage: https://www.gap-system.org

Reimer Behrends Email: behrends@gmail.com

Vladimir Janjic Email: vj32@st-andrews.ac.uk

Copyright

Contents

Chapter 1

Tasks

1.1 Overview

Tasks provide mid- to high-level functionality for programmers to describe asynchronous workflows. A task is an asynchronously or synchronously executing job; functions exist to create tasks that are executed concurrently, on demand, or in the current thread; to wait for their completion, check their status, and retrieve any results.

Here is a simple example of sorting a list in the background:

Example

```
gap> task := RunTask(x -> SortedList(x), [3,2,1]);;
gap> WaitTask(task);
gap> TaskResult(task);
[ 1, 2, 3 ]
```

`RunTask` (1.2.1) dispatches a task to run in the background; a task is described by a function and zero or more arguments that are passed to `RunTask` (1.2.1). `WaitTask` (1.2.9) waits for the task to complete; and `TaskResult` returns the result of the task.

`TaskResult` (1.2.11) does an implicit `WaitTask` (1.2.9), so the second line above can actually be omitted:

Example

```
gap> task := RunTask(x -> SortedList(x), [3,2,1]);;
gap> TaskResult(task);
[ 1, 2, 3 ]
```

It is simple to run two tasks in parallel. Let's compute the factorial of 10000 by splitting the work between two tasks:

Example

```
gap> task1 := RunTask(Product, [1..5000]);;
gap> task2 := RunTask(Product, [5001..10000]);;
gap> TaskResult(task1) * TaskResult(task2) = Factorial(10000);
true
```

You can use `DelayTask` (1.2.3) to delay executing the task until its result is actually needed.

Example

```
gap> task1 := DelayTask(Product, [1..5000]);;
gap> task2 := DelayTask(Product, [5001..10000]);;
gap> WaitTask(task1, task2);
```

```
gap> TaskResult(task1) * TaskResult(task2) = Factorial(10000);
true
```

Note that WaitTask (1.2.9) is used here to start execution of both tasks; otherwise, task2 would not be started until TaskResult(task1) has been evaluated.

To start execution of a delayed task, you can also use ExecuteTask. This has no effect if a task has already been running.

For convenience, you can also use ImmediateTask (1.2.7) to execute a task synchronously (i.e., the task is started immediately and the call does not return until the task has completed).

Example

```
gap> task := ImmediateTask(x -> SortedList(x), [3,2,1]);;
gap> TaskResult(task);
[ 1, 2, 3 ]
```

This is indistinguishable from calling the function directly, but provides the same interface as normal tasks.

If e.g. you want to call a function only for its side-effects, it can be useful to ignore the result of a task. RunAsyncTask (1.2.4) provides the necessary functionality. Such a task cannot be waited for and its result (if any) is ignored.

Example

```
gap> RunAsyncTask(function() Print("Hello, world!\n"); end);;
gap> !list
--- Thread 0 [0]
--- Thread 5 [5] (pending output)
gap> !5
--- Switching to thread 5
[5] Hello, world!
!0
--- Switching to thread 0
gap>
```

For more information on the multi-threaded user interface, see Chapter 4.

Task arguments are generally copied so that both the task that created them and the task that uses them can access the data concurrently without fear of race conditions. To avoid copying, arguments should be made shared or public (see the relevant parts of section 3.6 on migrating objects between regions); shared and public arguments will not be copied.

HPC-GAP currently has multiple implementations of the task API. To use an alternative implementation to the one documented here, set the environment variable GAP_WORKSTEALING to a non-empty value before starting GAP.

1.2 Running tasks

1.2.1 RunTask

▷ RunTask(*func[, arg1, ..., argn]*) (function)

RunTask prepares a task for execution and starts it. The task will call the function *func* with arguments *arg1* through *argn* (if provided). The return value of *func* is the result of the task. The RunTask call itself returns a task object that can be used by functions that expect a task argument.

1.2.2 ScheduleTask

▷ ScheduleTask(*condition, func[, arg1, ..., argn]*) (function)

ScheduleTask prepares a task for execution, but, unlike RunTask (1.2.1) does not start it until *condition* is met. See on how to construct conditions. Simple examples of conditions are individual tasks, where execution occurs after the task completes, or lists of tasks, where execution occurs after all tasks in the list complete.

Example

```
gap> t1 := RunTask(x->x*x, 3);;
gap> t2 := RunTask(x->x*x, 4);;
gap> t := ScheduleTask([t1, t2], function()
>           return TaskResult(t1) + TaskResult(t2);
>    end);;
gap> TaskResult(t);
25
```

While the above example could also be achieved with RunTask (1.2.1) in lieu of ScheduleTask, since TaskResult (1.2.11) would wait for *t1* and *t2* to complete, the above implementation does not actually start the final task until the others are complete, making it more efficient, since no additional worker thread needs to be occupied.

1.2.3 DelayTask

▷ DelayTask(*func[, arg1, ..., argn]*) (function)

DelayTask works as RunTask (1.2.1), but its start is delayed until it is being waited for (including implicitly by calling TaskResult (1.2.11)).

1.2.4 RunAsyncTask

▷ RunAsyncTask(*func[, arg1, ..., argn]*) (function)

RunAsyncTask creates an asynchronous task. It works like RunTask (1.2.1), except that its result will be ignored.

1.2.5 ScheduleAsyncTask

▷ ScheduleAsyncTask(*condition, func[, arg1, ..., argn]*) (function)

ScheduleAsyncTask creates an asynchronous task. It works like ScheduleTask (1.2.2), except that its result will be ignored.

1.2.6 MakeTaskAsync

▷ MakeTaskAsync(*task*) (function)

MakeTaskAsync turns a synchronous task into an asynchronous task that cannot be waited for and whose result will be ignored.

1.2.7 ImmediateTask

▷ ImmediateTask(*func[, arg1, ..., argn]*) (function)

ImmediateTask executes the task specified by its arguments synchronously, usually within the current thread.

1.2.8 ExecuteTask

▷ ExecuteTask(*task*) (function)

ExecuteTask starts *task* if it is not already running. It has only an effect if its argument is a task returned by DelayTask (1.2.3); otherwise, it is a no-op.

1.2.9 WaitTask

▷ WaitTask(*task1, ..., taskn*) (function)
▷ WaitTask(*condition*) (function)
▷ WaitTasks(*task1, ..., taskn*) (function)

WaitTask waits until *task1* through *taskn* have completed; after that, it returns. Alternatively, a condition can be passed to WaitTask in order to wait until a condition is met. See on how to construct conditions. WaitTasks is an alias for WaitTask.

1.2.10 WaitAnyTask

▷ WaitAnyTask(*task1, ..., taskn*) (function)

The WaitAnyTask function waits for any of its arguments to finish, then returns the number of that task.

Example

```
gap> task1 := DelayTask(x->SortedList(x), [3,2,1]);;
gap> task2 := DelayTask(x->SortedList(x), [6,5,4]);;
gap> which := WaitAnyTask(task1, task2);
2
gap> if which = 1 then
>      Display(TaskResult(task1));Display(TaskResult(task2));
>    else
>      Display(TaskResult(task2));Display(TaskResult(task1));
>    fi;
[ 4, 5, 6 ]
[ 1, 2, 3 ]
```

One can pass a list of tasks to WaitAnyTask as an argument; WaitAnyTask([task1, ..., taskn]) behaves identically to WaitAnyTask(task1, ..., taskn).

1.2.11 TaskResult

▷ TaskResult(*task*) (function)

The TaskResult function returns the result of a task. It implicitly calls WaitTask (1.2.9) if that is necessary. Multiple invocations of TaskResult with the same task argument will not do repeated waits and always return the same value.

If the function executed by *task* encounters an error, TaskResult returns fail. Whether *task* encountered an error can be checked via TaskSuccess (1.3.1). In case of an error, the error message can be retrieved via TaskError (1.3.2).

1.2.12 CullIdleTasks

▷ CullIdleTasks() (function)

This function terminates unused worker threads.

1.3 Information about tasks

1.3.1 TaskSuccess

▷ TaskSuccess(*task*) (function)

TaskSuccess waits for *task* and returns true if the it finished without encountering an error. Otherwise the function returns false.

1.3.2 TaskError

▷ TaskError(*task*) (function)

TaskError waits for *task* and returns its error message, if it encountered an error. If it did not encounter an error, the function returns fail.

1.3.3 CurrentTask

▷ CurrentTask() (function)

The CurrentTask returns the currently running task.

1.3.4 RunningTasks

▷ RunningTasks() (function)

This function returns the number of currently running tasks. Note that it is only an approximation and can change as new tasks are being started by other threads.

1.3.5 TaskStarted

▷ TaskStarted(*task*) (function)

This function returns true if the task has started executing (i.e., for any non-delayed task), false otherwise.

1.3.6 TaskFinished

▷ `TaskFinished(task)` (function)

This function returns true if the task has finished executing and its result is available, false otherwise.

1.3.7 TaskIsAsync

▷ `TaskIsAsync(task)` (function)

This function returns true if the task is asynchronous, true otherwise.

1.4 Cancelling tasks

HPC-GAP uses a cooperative model for task cancellation. A programmer can request the cancellation of another task, but it is up to that other task to actually terminate itself. The tasks library has functions to request cancellation, to test for the cancellation state of a task, and to perform actions in response to cancellation requests.

1.4.1 CancelTask

▷ `CancelTask(task)` (function)

`CancelTask` submits a request that *task* is to be cancelled.

1.4.2 TaskCancellationRequested

▷ `TaskCancellationRequested(task)` (function)

`TaskCancellationRequested` returns true if `CancelTask` (1.4.1) has been called for *task*, false otherwise.

1.4.3 OnTaskCancellation

▷ `OnTaskCancellation(exit_func)` (function)

`OnTaskCancellation` tests if cancellation for the current task has been requested. If so, then *exit_func* will be called (as a parameterless function) and the current task will be aborted. The result of the current task will be the value of *exit_func()*.

Example

```
gap> task := RunTask(function()
>      while true do
>        OnTaskCancellation(function() return 314; end);
>      od;
>    end);;
gap> CancelTask(task);
gap> TaskResult(task);
314
```

1.4.4 OnTaskCancellationReturn

▷ `OnTaskCancellationReturn(`*value*`)` (function)

`OnTaskCancellationReturn` is a convenience function that does the same as: `OnTaskCancellation(function() return value; end);`

1.5 Conditions

`ScheduleTask` (1.2.2) and `WaitTask` (1.2.9) can be made to wait on more complex conditions than just tasks. A condition is either a milestone, a task, or a list of milestones and tasks. `ScheduleTask` (1.2.2) starts its task and `WaitTask` (1.2.9) returns when the condition has been met. A condition represented by a task is met when the task has completed. A condition represented by a milestone is met when the milestone has been achieved (see below). A condition represented by a list is met when all conditions in the list have been met.

1.6 Milestones

Milestones are a way to represent abstract conditions to which multiple tasks can contribute.

1.6.1 NewMilestone

▷ `NewMilestone(`*[list]*`)` (function)

The `NewMilestone` function creates a new milestone. Its argument is a list of targets, which must be a list of integers and/or strings. If omitted, the list defaults to `[0]`.

1.6.2 ContributeToMilestone

▷ `ContributeToMilestone(`*milestone, target*`)` (function)

The `ContributeToMilestone` milestone function contributes the specified target to the milestone. Once all targets have been contributed to a milestone, it has been achieved.

1.6.3 AchieveMilestone

▷ `AchieveMilestone(`*milestone*`)` (function)

The `AchieveMilestone` function allows a program to achieve a milestone in a single step without adding individual targets to it. This is most useful in conjunction with the default value for `NewMilestone` (1.6.1), e.g.

Example

```
gap> m := NewMilestone();;
gap> AchieveMilestone(m);
```

>

1.6.4 IsMilestoneAchieved

▷ IsMilestoneAchieved(*milestone*) (function)

IsMilestoneAchieved tests explicitly if a milestone has been achieved. It returns true on success, false otherwise.

Example

```
gap> m := NewMilestone([1,2]);;
gap> ContributeToMilestone(m, 1);
gap> IsMilestoneAchieved(m);
false
gap> ContributeToMilestone(m, 2);
gap> IsMilestoneAchieved(m);
true
```

Chapter 2

Variables in HPC-GAP

Variables with global scope have revised semantics in HPC-GAP in order to address concurrency issues. The normal semantics of global variables that are only accessed by a single thread remain unaltered.

2.1 Global variables

Global variables in HPC-GAP can be accessed by all threads concurrently without explicit synchronization. Concurrent access is safe, but it is not deterministic. If multiple threads attempt to modify the same global variable simultaneously, the resulting value of the variable is random; it will be one of the values assigned by a thread, but it is impossible to predict with certainty which specific one will be assigned.

2.2 Thread-local variables

HPC-GAP supports the notion of thread-local variables. Thread-local variables are (after being declared as such) accessed and modified like global variables. However, unlike global variables, each thread can assign a distinct value to a thread-local variable.

Example

```
gap> MakeThreadLocal("x");
gap> x := 1;;
gap> WaitTask(RunTask(function() x := 2; end));
gap> x;
1
```

As can be seen here, the assignment to `x` in a separate thread does not overwrite the value of `x` in the main thread.

2.2.1 MakeThreadLocal

▷ `MakeThreadLocal(`*name*`)` (function)

`MakeThreadLocal` makes the variable described by the string *name* a thread-local variable. It normally does not give it an initial value; either explicit per-thread assignment or a call to

BindThreadLocal (2.2.2) or BindThreadLocalConstructor (2.2.3) to provide a default value is necessary.

If a global variable with the same name exists and is bound at the time of the call, its value will be used as the default value as though BindThreadLocal (2.2.2) had been called with that value as its second argument.

2.2.2 BindThreadLocal

▷ BindThreadLocal(*name*, *obj*) (function)

BindThreadLocal gives the thread-local variable described by the string *name* the default value *obj*. The first time the thread-local variable is accessed in a thread thereafter, it will yield *obj* as its value if it hasn't been assigned a specific value yet.

2.2.3 BindThreadLocalConstructor

▷ BindThreadLocalConstructor(*name*, *func*) (function)

BindThreadLocal (2.2.2) gives the thread-local variable described by the string *name* the constructor *func*. The first time the thread-local variable is accessed in a thread thereafter, it will yield *func()* as its value if it hasn't been assigned a specific value yet.

2.2.4 ThreadVar

▷ ThreadVar (global variable)

All thread-local variables are stored in the thread-local record ThreadVar. Thus, if x is a thread-local variable, using ThreadVar.x is the same as using x.

Chapter 3

How HPC-GAP organizes shared memory: Regions

HPC-GAP allows multiple threads to access data shared between them; to avoid common concurrency errors, such as race conditions, it partitions GAP objects into regions. Access to regions is regulated so that no two threads can modify objects in the same region at the same time and so that objects that are being read by one thread cannot concurrently be modified by another.

3.1 Thread-local regions

Each thread has an associated thread-local region. When a thread implicitly or explicitly creates a new object, that object initially belongs to the thread's thread-local region.

Only the thread can read or modify objects in its thread-local region. For other threads to access an object, that object has to be migrated into a different region first.

3.2 Shared regions

Shared regions are explicitly created through the `ShareObj` (3.9.9) and `ShareSingleObj` (3.9.15) primitives (see below). Multiple threads can access them concurrently, but accessing them requires that a thread uses an `atomic` statement to acquire a read or write lock beforehand.

See the section on `atomic` statements (3.9.43) for details.

3.3 Ordering of shared regions

Shared regions are by default ordered; each shared region has an associated numeric precedence level. Regions can generally only be locked in order of descending precedence. The purpose of this mechanism is to avoid accidental deadlocks.

The ordering requirement can be overridden in two ways: regions with a negative precedence are excluded from it. This exception should be used with care, as it can lead to deadlocks.

Alternatively, two or more regions can be locked simultaneously via the `atomic` statement. In this case, the ordering of these regions relative to each other can be arbitrary.

3.4 The public region

A special public region contains objects that only permit atomic operations. These include, in particular, all immutable objects (immutable in the sense that their in-memory representation cannot change).

All threads can access objects in the public region at all times without needing to acquire a read- or write-lock beforehand.

3.5 The read-only region

The read-only region is another special region that contains objects that are only meant to be read; attempting to modify an object in that region will result in a runtime error. To obtain a modifiable copy of such an object, the `CopyRegion` (3.9.29) primitive can be used.

3.6 Migrating objects between regions

Objects can be migrated between regions using a number of functions. In order to migrate an object, the current thread must have exclusive access to that object; the object must be in its thread-local region or it must be in a shared region for which the current thread holds a write lock.

The `ShareObj` (3.9.9) and `ShareSingleObj` (3.9.15) functions create a new shared region and migrate their respective argument to that region; `ShareObj` will also migrate all subobjects that are within the same region, while `ShareSingleObj` will leave the subobjects unaffected.

The `MigrateObj` (3.9.21) and `MigrateSingleObj` (3.9.22) functions migrate objects to an existing region. The first argument of either function is the object to be migrated; the second is either a region (as returned by the `RegionOf` (3.9.7) function) or an object whose containing region the first argument is to be migrated to.

The current thread needs exclusive access to the target region (denoted by the second argument) for the operation to succeed. If successful, the first argument will be in the same region as the second argument afterwards. In the case of `MigrateObj` (3.9.21), all subobjects within the same region as the first argument will also be migrated to the target region.

Finally, `AdoptObj` (3.9.26) and `AdoptSingleObj` (3.9.27) are special cases of `MigrateObj` (3.9.21) and `MigrateSingleObj` (3.9.22), where the target region is the thread-local region of the current thread.

To migrate objects to the read-only region, one can use `MakeReadOnlyObj` (3.9.35) and `MakeReadOnlySingleObj` (3.9.36). The first migrates its argument and all its subjobjects that are within the same region to the read-only region; the second migrates only the argument itself, but not its subobjects.

It is generally not possible to migrate objects explicitly to the public region; only objects with purely atomic operations can be made public and that is done automatically when they are created.

The exception are immutable objects. When `MakeImmutable` (**Reference: MakeImmutable**) is used, its argument is automatically moved to the public region.

Example

```
gap> RegionOf(MakeImmutable([1,2,3]));
<public region>
```

3.7 Region names

Regions can be given names, either explicitly via SetRegionName (3.9.38) or when they are created via ShareObj (3.9.9) and ShareSingleObj (3.9.15). Thread-local regions, the public, and the read-only region are given names by default.

Multiple regions can have the same name.

3.8 Controlling access to regions

If either GAP code or a kernel primitive attempts to access an object that it is not allowed to access according to these semantics, either a "write guard error" (for a failed write access) or a "read guard error" (for a failed read access) will be raised. The global variable LastInaccessible will contain the object that caused such an error.

One exception is that threads can modify objects in regions that they have only read access (but not write access) to using write-once functions - see section 3.11.

To inspect objects whose contents lie in other regions (and therefore cannot be displayed by PrintObj (**Reference: PrintObj**) or ViewObj (**Reference: ViewObj**), the functions ViewShared (3.9.41) and UNSAFE_VIEW (3.9.42) can be used.

3.9 Functions relating to regions

3.9.1 NewRegion

▷ NewRegion(*[name,]prec*) (function)

The function NewRegion creates a new shared region. If the optional argument *name* is provided, then the name of the new region will be set to *name*.

Example

```
gap> NewRegion("example region");
<region: example region>
```

NewRegion will create a region with a high precedence level. It is intended to be called by user code. The exact precedence level can be adjusted with *prec*, which must be an integer in the range [-1000..1000]; *prec* will be added to the normal precedence level.

3.9.2 NewLibraryRegion

▷ NewLibraryRegion(*[name,]prec*) (function)

NewLibraryRegion functions like NewRegion (3.9.1), except that the precedence of the region it creates is below that of NewRegion (3.9.1). It is intended to be used by user libraries and GAP packages.

3.9.3 NewSystemRegion

▷ NewSystemRegion(*[name,]prec*) (function)

NewSystemRegion functions like NewRegion (3.9.1), except that the precedence of the region it creates is below that of NewLibraryRegion (3.9.2). It is intended to be used by the standard GAP library.

3.9.4 NewKernelRegion

▷ NewKernelRegion([name,]prec) (function)

NewKernelRegion functions like NewRegion (3.9.1), except that the precedence of the region it creates is below that of NewSystemRegion (3.9.3). It is intended to be used by the GAP kernel, and GAP library code that interacts closely with the kernel.

3.9.5 NewInternalRegion

▷ NewInternalRegion([name]) (function)

NewInternalRegion functions like NewRegion (3.9.1), except that the precedence of the region it creates is the lowest available. It is intended to be used for regions that are self-contained; i.e. no function that uses such a region may lock another region while accessing it. The precedence level of an internal region cannot be adjusted.

3.9.6 NewSpecialRegion

▷ NewSpecialRegion([name]) (function)

NewLibraryRegion (3.9.2) functions like NewRegion (3.9.1), except that the precedence of the region it creates is negative. It is thus exempt from normal ordering and deadlock checks.

3.9.7 RegionOf

▷ RegionOf(obj) (function)

Example

```
gap> RegionOf(1/2);
<public region>
gap> RegionOf([1,2,3]);
<region: thread region #0>
gap> RegionOf(ShareObj([1,2,3]));
<region 0x45deaa0>
gap> RegionOf(ShareObj([1,2,3]));
<region 0x45deaa0>
gap> RegionOf(ShareObj([1,2,3], "test region"));
<region: test region>
```

Note that the unique number that each region is identified with is system-specific and can change each time the code is being run. Region objects returned by RegionOf can be compared:

Example

```
gap> RegionOf([1,2,3]) = RegionOf([4,5,6]);
true
```

The result in this example is true because both lists are in the same thread-local region.

3.9.8 RegionPrecedence

▷ `RegionPrecedence(obj)` (function)

`RegionPrecedence` will return the precedence of the region of *obj*.

Example

```
gap> RegionPrecedence(NewRegion("Test"));
30000
gap> RegionPrecedence(NewRegion("Test2", 1));
30001
gap> RegionPrecedence(NewLibraryRegion("LibTest", -1));
19999
```

3.9.9 ShareObj

▷ `ShareObj(obj[[, name], prec])` (function)

The `ShareObj` function creates a new shared region and migrates the object and all its subobjects to that region. If the optional argument *name* is provided, then the name of the new region is set to *name*.

`ShareObj` will create a region with a high precedence level. It is intended to be called by user code. The actual precedence level can be adjusted by the optional *prec* argument in the same way as for `NewRegion` (3.9.1).

3.9.10 ShareLibraryObj

▷ `ShareLibraryObj(obj[[, name], prec])` (function)

`ShareLibraryObj` functions like `ShareObj` (3.9.9), except that the precedence of the region it creates is below that of `ShareObj` (3.9.9). It is intended to be used by user libraries and GAP packages.

3.9.11 ShareSystemObj

▷ `ShareSystemObj(obj[[, name], prec])` (function)

`ShareSystemObj` functions like `ShareObj` (3.9.9), except that the precedence of the region it creates is below that of `ShareLibraryObj` (3.9.10). It is intended to be used by the standard GAP library.

3.9.12 ShareKernelObj

▷ `ShareKernelObj(obj[[, name], prec])` (function)

`ShareKernelObj` functions like `ShareObj` (3.9.9), except that the precedence of the region it creates is below that of `ShareSystemObj` (3.9.11). It is intended to be used by the GAP kernel, and GAP library code that interacts closely with the kernel.

3.9.13 ShareInternalObj

▷ ShareInternalObj(*obj[, name]*) (function)

ShareInternalObj functions like ShareObj (3.9.9), except that the precedence of the region it creates is the lowest available. It is intended to be used for regions that are self-contained; i.e. no function that uses such a region may lock another region while accessing it.

3.9.14 ShareSpecialObj

▷ ShareSpecialObj(*obj[, name]*) (function)

ShareLibraryObj (3.9.10) functions like ShareObj (3.9.9), except that the precedence of the region it creates is negative. It is thus exempt from normal ordering and deadlock checks.

3.9.15 ShareSingleObj

▷ ShareSingleObj(*obj[[, name], prec]*) (function)

The ShareSingleObj function creates a new shared region and migrates the object, but not its subobjects, to that region. If the optional argument *name* is provided, then the name of the new region is set to *name*.

Example

```
gap> m := [ [1, 2], [3, 4] ];;
gap> ShareSingleObj(m);;
gap> atomic readonly m do
>      Display([ IsShared(m), IsShared(m[1]), IsShared(m[2]) ]);
>    od;
[ true, false, false ]
```

ShareSingleObj will create a region with a high precedence level. It is intended to be called by user code. The actual precedence level can be adjusted by the optional *prec* argument in the same way as for NewRegion (3.9.1).

3.9.16 ShareSingleLibraryObj

▷ ShareSingleLibraryObj(*obj[[, name], prec]*) (function)

ShareSingleLibraryObj functions like ShareSingleObj (3.9.15), except that the precedence of the region it creates is below that of ShareSingleObj (3.9.15). It is intended to be used by user libraries and GAP packages.

3.9.17 ShareSingleSystemObj

▷ ShareSingleSystemObj(*obj[[, name], prec]*) (function)

ShareSingleSystemObj functions like ShareSingleObj (3.9.15), except that the precedence of the region it creates is below that of ShareSingleLibraryObj (3.9.16). It is intended to be used by the standard GAP library.

3.9.18 ShareSingleKernelObj

▷ ShareSingleKernelObj(*obj[[, name], prec]*) (function)

ShareSingleKernelObj functions like ShareSingleObj (3.9.15), except that the precedence of the region it creates is below that of ShareSingleSystemObj (3.9.17). It is intended to be used by the GAP kernel, and GAP library code that interacts closely with the kernel.

3.9.19 ShareSingleInternalObj

▷ ShareSingleInternalObj(*obj[, name]*) (function)

ShareSingleInternalObj functions like ShareSingleObj (3.9.15), except that the precedence of the region it creates is the lowest available. It is intended to be used for regions that are self-contained; i.e. no function that uses such a region may lock another region while accessing it.

3.9.20 ShareSingleSpecialObj

▷ ShareSingleSpecialObj(*obj[, name]*) (function)

ShareSingleLibraryObj (3.9.16) functions like ShareSingleObj (3.9.15), except that the precedence of the region it creates is negative. It is thus exempt from normal ordering and deadlock checks.

3.9.21 MigrateObj

▷ MigrateObj(*obj, target*) (function)

The MigrateObj function migrates *obj* (and all subobjects contained within the same region) to the region denoted by the *target* argument. Here, *target* can either be a region object returned by RegionOf (3.9.7) or a normal gap object. If *target* is a normal gap object, *obj* will be migrated to the region containing target.

For the operation to succeed, the current thread must have exclusive access to the target region and the object being migrated.

3.9.22 MigrateSingleObj

▷ MigrateSingleObj(*obj, target*) (function)

The MigrateSingleObj function works like MigrateObj (3.9.21), except that it does not migrate the subobjects of obj.

3.9.23 LockAndMigrateObj

▷ LockAndMigrateObj(*obj, target*) (function)

The LockAndMigrateObj function works like MigrateObj (3.9.21), except that it will automatically try to acquire a lock for the region containing *target* if it does not have one already.

3.9.24 IncorporateObj

▷ IncorporateObj(*target, index, value*) (function)

The IncorporateObj function allows convenient migration to a shared list or record. If *target* is a list, then IncorporateObj is equivalent to:

Example

```
IncorporateObj := function(target, index, value)
  atomic value do
    target[index] := MigrateObj(value, target)
  od;
end;
```

If *target* is a record, then it is equivalent to:

Example

```
IncorporateObj := function(target, index, value)
  atomic value do
    target.(index) := MigrateObj(value, target)
  od;
end;
```

The intended purpose is the population of a shared list or record with values after its creation. Example:

Example

```
gap> list := ShareObj([]);
gap> atomic list do
>      IncorporateObj(list, 1, [1,2,3]);
>      IncorporateObj(list, 2, [4,5,6]);
>      IncorporateObj(list, 3, [7,8,9]);
>    od;
gap> ViewShared(list);
[ [ 1, 2, 3 ], [ 4, 5, 6 ], [ 7, 8, 9 ] ]
```

Using plain assignment would leave the newly created lists in the thread-local region.

3.9.25 AtomicIncorporateObj

▷ AtomicIncorporateObj(*target, index, value*) (function)

AtomicIncorporateObj extends IncorporateObj (3.9.24) by also locking the target. I.e., for a list, it is equivalent to:

Example

```
AtomicIncorporateObj := function(target, index, value)
  atomic target, value do
    target[index] := MigrateObj(value, target)
  od;
end;
```

If target is a record, then it is equivalent to:

Example

```
AtomicIncorporateObj := function(target, index, value)
  atomic value do
    target.(index) := MigrateObj(value, target)
  od;
end;
```

3.9.26 AdoptObj

▷ AdoptObj(*obj*) (function)

The AdoptObj function migrates *obj* (and all its subobjects contained within the same region) to the thread's current region. It requires exclusive access to *obj*.

Example

```
gap> l := ShareObj([1,2,3]);;
gap> IsThreadLocal(l);
false
gap> atomic l do AdoptObj(l); od;
gap> IsThreadLocal(l);
true
```

3.9.27 AdoptSingleObj

▷ AdoptSingleObj(*obj*) (function)

The AdoptSingleObj function works like AdoptObj (3.9.26), except that it does not migrate the subobjects of *obj*.

3.9.28 LockAndAdoptObj

▷ LockAndAdoptObj(*obj*) (function)

The LockAndAdoptObj function works like AdoptObj (3.9.26), except that it will attempt acquire an exclusive lock for the region containing *obj* if it does not have one already.

3.9.29 CopyRegion

▷ CopyRegion(*obj*) (function)

The CopyRegion function performs a structural copy of *obj*. The resulting objects will be located in the current thread's thread-local region. The function returns the copy as its result.

Example

```
gap> l := MakeReadOnlyObj([1,2,3]);
[ 1, 2, 3 ]
gap> l2 := CopyRegion(l);
[ 1, 2, 3 ]
gap> RegionOf(l) = RegionOf(l2);
false
gap> IsIdenticalObj(l, l2);
```

```
false
gap> l = l2;
true
```

3.9.30 IsPublic

▷ IsPublic(*obj*) (function)

The IsPublic function returns true if its argument is an object in the public region, false otherwise.

Example

```
gap> IsPublic(1/2);
true
gap> IsPublic([1,2,3]);
false
gap> IsPublic(ShareObj([1,2,3]));
false
gap> IsPublic(MakeImmutable([1,2,3]));
true
```

3.9.31 IsThreadLocal

▷ IsThreadLocal(*obj*) (function)

The IsThreadLocal function returns true if its argument is an object in the current thread's thread-local region, false otherwise.

Example

```
gap> IsThreadLocal([1,2,3]);
true
gap> IsThreadLocal(ShareObj([1,2,3]));
false
gap> IsThreadLocal(1/2);
false
gap> RegionOf(1/2);
<public region>
```

3.9.32 IsShared

▷ IsShared(*obj*) (function)

The IsShared function returns true if its argument is an object in a shared region. Note that if the current thread does not hold a lock on that shared region, another thread can migrate *obj* to a different region before the result is being evaluated; this can lead to race conditions. The function is intended primarily for debugging, not to build actual program logic around.

3.9.33 HaveReadAccess

▷ HaveReadAccess(*obj*) (function)

The HaveReadAccess function returns true if the current thread has read access to *obj*.

Example

```
gap> HaveReadAccess([1,2,3]);
true
gap> l := ShareObj([1,2,3]);;
gap> HaveReadAccess(l);
false
gap> atomic readonly l do t := HaveReadAccess(l); od;; t;
true
```

3.9.34 HaveWriteAccess

▷ HaveWriteAccess(*obj*) (function)

The HaveWriteAccess function returns true if the current thread has write access to *obj*.

Example

```
gap> HaveWriteAccess([1,2,3]);
true
gap> l := ShareObj([1,2,3]);;
gap> HaveWriteAccess(l);
false
gap> atomic readwrite l do t := HaveWriteAccess(l); od;; t;
true
```

3.9.35 MakeReadOnlyObj

▷ MakeReadOnlyObj(*obj*) (function)

The MakeReadOnlyObj function migrates *obj* and all its subobjects that are within the same region as *obj* to the read-only region. It returns *obj*.

3.9.36 MakeReadOnlySingleObj

▷ MakeReadOnlySingleObj(*obj*) (function)

The MakeReadOnlySingleObj function migrates *obj*, but not any of its subobjects, to the read-only region. It returns *obj*.

3.9.37 IsReadOnlyObj

▷ IsReadOnlyObj(*obj*) (function)

The IsReadOnlyObj function returns true if *obj* is in the read-only region, false otherwise.

Example

```
gap> IsReadOnlyObj([1,2,3]);
false
gap> IsReadOnlyObj(MakeImmutable([1,2,3]));
false
gap> IsReadOnlyObj(MakeReadOnlyObj([1,2,3]));
true
```

3.9.38 SetRegionName

▷ SetRegionName(*obj*, *name*) (function)

The SetRegionName function sets the name of the region of *obj* to *name*.

3.9.39 ClearRegionName

▷ ClearRegionName(*obj*) (function)

The ClearRegionName function clears the name of the region of *obj* to *name*.

3.9.40 RegionName

▷ RegionName(*obj*) (function)

The RegionName function returns the name of the region of *obj*. If that region does not have a name, fail will be returned.

3.9.41 ViewShared

▷ ViewShared(*obj*) (function)

The ViewShared function allows the inspection of objects in shared regions. It will try to lock the region and then call ViewObj(obj). If it cannot acquire a lock for the region, it will simply display the normal description of the object.

3.9.42 UNSAFE_VIEW

▷ UNSAFE_VIEW(*obj*) (function)

The UNSAFE_VIEW (3.9.42) function allows the inspection of any object in the system, regardless of whether the current thread has access to the region containing it. It should be used with care: If the object inspected is being modified by another thread concurrently, the resulting behavior is undefined.

Moreover, the function works by temporarily disabling read and write guards for regions, so other threads may corrupt memory rather than producing errors.

It is generally safe to use if all threads but the current one are paused.

3.9.43 The atomic statement.

The atomic statement ensures exclusive or read-only access to one or more shared regions for statements within its scope. It has the following syntax:

Example

```
atomic ([readwrite|readonly] expr (, expr)* )* do
  statements
od;
```

Each expression is evaluated and the region containing the resulting object is locked with either a read-write or read-only lock, depending on the keyword preceding the expression. If neither the `readwrite` nor the `readonly` keyword was provided, read-write locks are used by default. Examples:

Example

```
gap> l := ShareObj([1,2,3]);;
gap> atomic readwrite l do l[3] := 9; od;
gap> atomic l do l[2] := 4; od;
gap> atomic readonly l do Display(l); od;
[ 1, 4, 9 ]
```

Example

```
gap> l := ShareObj([1,2,3,4,5]);;
gap> l2 := ShareObj([6,7,8]);;
gap> atomic readwrite l, readonly l2 do
>      for i in [1..3] do l[i] := l2[i]; od;
>      l3 := AdoptObj(l);
>    od;
gap> l3;
[ 6, 7, 8, 4, 5 ]
```

Atomic statements must observe region ordering. That means that the highest precedence level of a region locked by an atomic statement must be less than the lowest precedene level of a region that is locked by the same thread at the time the atomic statement is executed.

3.10 Atomic functions

Instead of atomic regions, entire functions can be declared to be atomic. This has the same effect as though the function's body were enclosed in an atomic statement. Function arguments can be declared either `readwrite` or `readonly`; they will be locked in the same way as for a lock statement. If a function argument is preceded by neither `readwrite` nor `readonly`, the corresponding object will not be locked. Example:

Example

```
gap> AddAtomic := atomic function(readwrite list, readonly item)
>      Add(list, item);
>    end;
```

3.11 Write-once functionality

There is an exception to the rule that objects can only be modified if a thread has write access to a region. A limited sets of objects can be modified using the "bind once" family of functions. These allow the modifications of objects to which a thread has read access in a limited fashion.

For reasons of implementation symmetry, these functions can also be used on the atomic versions of these objects.

Implementation note: The functionality is not currently available for component objects.

3.11.1 BindOnce

▷ `BindOnce(`*obj, index, value*`)` (function)

`BindOnce` modifies *obj*, which can be a positional object, atomic positional object, component object, or atomic component object. It inspects `obj![index]` for the positional versions or `obj!.(index)` for the component versions. If the respective element is not yet bound, *value* is assigned to that element. Otherwise, no modification happens. The test and modification occur as one atomic step. The function returns the value of the element; i.e. the old value if the element was bound and *value* if it was unbound.

The intent of this function is to allow concurrent initialization of objects, where multiple threads may attempt to set a value concurrently. Only one will succeed; all threads can then use the return value of `BindOnce` as the definitive value of the element. It also allows for the lazy initialization of objects in the read-only region.

The current thread needs to have at least read access to *obj*, but does not require write access.

3.11.2 TestBindOnce

▷ `TestBindOnce(`*obj, index, value*`)` (function)

`TestBindOnce` works like `BindOnce` (3.11.1), except that it returns `true` if the value could be bound and `false` otherwise.

3.11.3 BindOnceExpr

▷ `BindOnceExpr(`*obj, index, expr*`)` (function)

`BindOnceExpr` works like `BindOnce` (3.11.1), except that it evaluates the parameterless function *expr* to determine the value. It will only evaluate *expr* if the element is not bound.

For positional objects, the implementation works as follows:

Example

```
BindOnceExprPosObj := function(obj, index, expr)
  if not IsBound(obj![index]) then
    return BindOnce(obj, index, expr());
  else
    return obj![index];
  fi;
end;
```

The implementation for component objects works analogously.

The intent is to avoid unnecessary computations if the value is already bound. Note that this cannot be avoided entirely, because `obj![index]` or `obj!.(index)` can be bound while *expr* is evaluated, but it can minimize such occurrences.

3.11.4 TestBindOnceExpr

▷ `TestBindOnceExpr(`*obj, index, expr*`)` (function)

`TestBindOnceExpr` works like `BindOnceExpr` (3.11.3), except that it returns `true` if the value could be bound and `false` otherwise.

3.11.5 StrictBindOnce

▷ `StrictBindOnce(`*obj, index, expr*`)` (function)

`StrictBindOnce` works like `BindOnce` (3.11.1), except that it raises an error if the element is already bound. This is intended for cases where a read-only object is initialized, but where another thread trying to initialize it concurrently would be an error.

Chapter 4

Console User Interface

HPC-GAP has a multi-threaded user interface to assist with the development and debugging of concurrent programs. This user interface is enabled by default; to disable it, and use the single-threaded interface, GAP has to be started with the `-S` option.

4.1 Console UI commands

The console user interface provides the user with the option to control threads by commands prefixed with an exclamation mark ("!"). Those commands are listed below.

For ease of use, users only need to type as many letters of each commands so that it can be unambiguously selected. Thus, the shell will recognize `!l` as an abbreviation for `!list`.

4.1.1 !shell [name]

Starts a new shell thread and switches to it. Optionally, a name for the thread can be provided.

Example

```
gap> !shell
--- Switching to thread 4
[4] gap>
```

4.1.2 !fork [name]

Starts a new background shell thread. Optionally, a name for the thread can be provided.

Example

```
gap> !fork
--- Created new thread 5
```

4.1.3 !list

List all current threads that are interacting with the user. This does not list threads created with `CreateThread()` that have not entered a break loop.

Example

```
gap> !list
--- Thread 0 [0]
--- Thread 4 [4]
--- Thread 5 [5] (pending output)
```

4.1.4 !kill id

Terminates the specified thread.

4.1.5 !break id

Makes the specified thread enter a break loop.

4.1.6 !name [id] name

Give the thread with the numerical identifier or name `id` the name `name`.

Example

```
gap> !name 5 test
gap> !list
--- Thread 0 [0]
--- Thread 4 [4]
--- Thread test [5] (pending output)
```

4.1.7 !info id

Provide information about the thread with the numerical identifier or name `id`. *Not yet implemented*.

4.1.8 !hide [id|*]

Hide output from the thread with the numerical identifier or name `id` when it is not the foreground thread. If no thread is specified, make this the default behavior for future threads.

4.1.9 !watch [id|*]

Show output from the thread with the numerical identifier or name `id` even when it is not the foreground thread. If no thread is specified, make this the default behavior for future threads.

4.1.10 !keep num

Keep `num` lines of output from each thread.

4.1.11 !prompt (id|*) string

Set the prompt for the specified thread (or for all newly created threads if `*` was specified) to be `string`. If the string contains the pattern `id`, it is replaced with the numerical id of the thread; if it contains the pattern `name`, it is replaced with the name of the thread; if the thread has no name, the numerical id is displayed instead.

4.1.12 !prefix (id|*) string

Prefix the output from the specified thread (or for all newly created threads if `*` was specified) wiht `string`. The same substitution rules as for the `!prompt` command apply.

4.1.13 !select id

Make the specified thread the foreground thread.

Example

```
gap> !select 4
gap> !select 4
--- Switching to thread 4
[4] gap>
```

4.1.14 !next

Make the next thread in numerical order the foreground thread.

4.1.15 !previous

Make the previous thread in numerical order the foreground thread.

4.1.16 !replay num [id]

Display the last `num` lines of output of the specified thread. If no thread was specified, display the last `num` lines of the current foreground thread.

4.1.17 !id

`!id` is a shortcut for `!select id`.

4.1.18 !source file

Read commands from file `file`.

4.1.19 !alias shortcut expansion

Create an alias. After defining the alias, `!shortcut 'rest of line'` will be replaced with `!expansion 'rest of line'`.

4.1.20 !unalias shortcut

Removes the specified alias.

4.1.21 !eval expr

Evaluates `expr` as a command.

4.1.22 !run function string

Calls the function with name `function`, passing it the single argument `string` as a GAP string.

4.2 GAP functions to access the Shell UI

There are several functions to access the basic functionality of the shell user interface. Other than `TextUIRegisterCommand` (4.2.1), they can only be called from within a registered command.

Threads can be specified either by their numerical identifier or by their name (as a string). The empty string can be used to specify the current foreground thread.

4.2.1 TextUIRegisterCommand

▷ `TextUIRegisterCommand(`*`name, func`*`)` (function)

Registers the command `!name` with the shell UI. It will call <func> with the rest of the command line passed as a string argument when typed.

4.2.2 TextUIForegroundThread

▷ `TextUIForegroundThread()` (function)

Returns the numerical identifier of the current foreground thread.

4.2.3 TextUIForegroundThreadName

▷ `TextUIForegroundThreadName()` (function)

Returns the name of the current foreground thread or `fail` if the current foreground thread has no name.

4.2.4 TextUISelectThread

▷ `TextUISelectThread(`*`id`*`)` (function)

Makes *`id`* the current foreground thread. Returns `true` or `false` to indicate success.

4.2.5 TextUIOutputHistory

▷ `TextUIOutputHistory(`*`id, count`*`)` (function)

Returns the last *`count`* lines of the thread specified by *`id`* (which can be a numerical identifier or a name). Returns `fail` if there is no such thread.

4.2.6 TextUISetOutputHistoryLength

▷ `TextUISetOutputHistoryLength(`*`length`*`)` (function)

By default, retain *`length`* lines of output history from each thread.

4.2.7 TextUINewSession

▷ `TextUINewSession(`*`foreground, name`*`)` (function)

Creates a new shell thread. Here, *`foreground`* is a boolean variable specifying whether it should be made the new foreground thread and *`name`* is the name of the thread. The empty string can be used to leave the thread without a name.

4.2.8 TextUIRunCommand

▷ `TextUIRunCommand(`*`command`*`)` (function)

Run the command denoted by *`command`* as though a user had typed it. The command must not contain a newline character.

4.2.9 TextUIWritePrompt

▷ `TextUIWritePrompt()` (function)

Display a prompt for the current thread.

Chapter 5

Atomic objects

HPC-GAP provides a number of atomic object types. These can be accessed by multiple threads concurrently without requiring explicit synchronization, but can have non-deterministic behavior for complex operations. Atomic lists are fixed-size lists; they can be assigned to and read from like normal plain lists. Atomic records are atomic versions of plain records. Unlike plain records, though, it is not possible to delete elements from an atomic record. The primary use of atomic lists and records is to facilitate storing the result of idempotent operations and to support certain low-level operations. Atomic lists and records can have three different replacement policies: write-once, strict write-once, and rewritable. The replacement policy determines whether an already assigned element can be changed. The write-once policy allows elements to be assigned only once, with subsequent assignments being ignored; the strict write-once policy allows elements also to be assigned only once, but subsequent assignments will raise an error; the rewritable policy allows elements to be assigned different values repeatedly. The default for new atomic objects is to be rewritable. Thread-local records are variants of plain records that are replicated on a per-thread basis.

5.1 Atomic lists

Atomic lists are created using the `AtomicList` or `FixedAtomicList` functions. After creation, they can be used exactly like any other list, except that atomic lists created with `FixedAtomicList` cannot be resized. Their contents can also be read as normal plain lists using `FromAtomicList`.

Example

```
gap> a := AtomicList([1,2,4]);
<atomic list of size 3>
gap> WaitTask(RunTask(function() a[1] := a[1] + a[2]; end));
gap> a[1];
3
gap> FromAtomicList(a);
[ 3, 2, 4 ]
```

Because multiple threads can read and write the list concurrently without synchronization, the results of modifying the list may be non-deterministic. It is faster to write to fixed atomic lists than to a resizable atomic list.

5.1.1 AtomicList

▷ AtomicList(*list*) (function)
▷ AtomicList(*count*, *obj*) (function)

AtomicList is used to create a new atomic list. It takes either a plain list as an argument, in which case it will create a new atomic list of the same size, populated by the same elements; or it takes a count and an object argument. In that case, it creates an atomic list with *count* elements, each set to the value of *obj*.

Example

```
gap> al := AtomicList([3, 1, 4]);
<atomic list of size 3>
gap> al[3];
4
gap> al := AtomicList(10, ‘"alpha");
<atomic list of size 10>
gap> al[3];
"alpha"
gap> WaitTask(RunTask(function() al[3] := ‘"beta"; end));
gap> al[3];
"beta"
```

5.1.2 FixedAtomicList

▷ FixedAtomicList(*list*) (function)
▷ FixedAtomicList(*count*, *obj*) (function)

FixedAtomicList works like AtomicList (5.1.1) except that the resulting list cannot be resized.

5.1.3 MakeFixedAtomicList

▷ MakeFixedAtomicList(*list*) (function)

MakeFixedAtomicList turns a resizable atomic list into a fixed atomic list.

Example

```
gap> a := AtomicList([99]);
<atomic list of size 1>
gap> a[2] := 100;
100
gap> MakeFixedAtomicList(a);
<fixed atomic list of size 2>
gap> a[3] := 101;
Error, Atomic List Element: <pos>=3 is an invalid index for <list>
```

5.1.4 FromAtomicList

▷ FromAtomicList(*atomic_list*) (function)

FromAtomicList returns a plain list containing the same elements as *atomic_list* at the time of the call. Because other threads can write concurrently to that list, the result is not guaranteed to be deterministic.

Example

```
gap> al := AtomicList([10, 20, 30]);;
gap> WaitTask(RunTask(function() al[2] := 40; end));
gap> FromAtomicList(al);
[ 10, 40, 30 ]
```

5.1.5 ATOMIC_ADDITION

▷ ATOMIC_ADDITION(*atomic_list, index, value*) (function)

ATOMIC_ADDITION (5.1.5) is a low-level operation that atomically adds *value* to *atomic_list[index]*. It returns the value of *atomic_list[index]* after the addition has been performed.

Example

```
gap> al := FixedAtomicList([4,5,6]);;
gap> ATOMIC_ADDITION(al, 2, 7);
12
gap> FromAtomicList(al);
[ 4, 12, 6 ]
```

5.1.6 COMPARE_AND_SWAP

▷ COMPARE_AND_SWAP(*atomic_list, index, old, new*) (function)

COMPARE_AND_SWAP (5.1.6) is an atomic operation. It atomically compares *atomic_list[index]* to *old* and, if they are identical, replaces the value (in the same atomic step) with *new*. It returns true if the replacement took place, false otherwise.

The primary use of COMPARE_AND_SWAP (5.1.6) is to implement certain concurrency primitives; most programmers will not need to use it.

5.2 Atomic records and component objects

Atomic records are atomic counterparts to plain records. They support assignment to individual record fields, and conversion to and from plain records.

Assignment semantics can be specified on a per-record basis if the assigned record field is already populated, allowing either an overwrite, keeping the existing value, or raising an error.

It is not possible to unbind atomic record elements.

Like plain records, atomic records can be converted to component objects using Objectify.

5.2.1 AtomicRecord

▷ AtomicRecord(*capacity*) (function)

▷ AtomicRecord(*record*) (function)

`AtomicRecord` is used to create a new atomic record. Its single optional argument is either a positive integer, denoting the intended capacity (i.e., number of elements to be held) of the record, in which case a new empty atomic record with that initial capacity will be created. Alternatively, the caller can provide a plain record with which to initially populate the atomic record.

Example

```
gap> r := AtomicRecord(rec( x := 2 ));
<atomic record 1/2 full>
gap> r.y := 3;
3
gap> TaskResult(RunTask(function() return r.x + r.y; end));
5
gap> [ r.x, r.y ];
[ 2, 3 ]
```

Any atomic record can later grow beyond its initial capacity. There is no limit to the number of elements it can hold other than available memory.

5.2.2 FromAtomicRecord

▷ `FromAtomicRecord(`*record*`)` (function)

`FromAtomicRecord` returns a plain record copy of the atomic record *record*. This copy is shallow; elements of *record* will not also be copied.

Example

```
gap> r := AtomicRecord();;
gap> r.x := 1;; r.y := 2;; r.z := 3;;
gap> FromAtomicRecord(r);
rec( x := 1, y := 2, z := 3 )
```

5.3 Replacement policy functions

There are three functions that set the replacement policy of an atomic object. All three can also be used with plain lists and records, in which case an atomic version of the list or record is first created. This allows programmers to elide `AtomicList` (5.1.1) and `AtomicRecord` (5.2.1) calls when the next step is to change their policy.

5.3.1 MakeWriteOnceAtomic

▷ `MakeWriteOnceAtomic(`*obj*`)` (function)

`MakeWriteOnceAtomic` takes a list, record, atomic list, atomic record, atomic positional object, or atomic component object as its argument. If the argument is a non-atomic list or record, then the function first creates an atomic copy of the argument. The function then changes the replacement policy of the object to write-once: if an element of the object is already bound, then further attempts to assign to it will be ignored.

5.3.2 MakeStrictWriteOnceAtomic

▷ MakeStrictWriteOnceAtomic(*obj*) (function)

MakeStrictWriteOnceAtomic works like MakeWriteOnceAtomic (5.3.1), except that the replacement policy is being changed to being strict write-once: if an element is already bound, then further attempts to assign to it will raise an error.

5.3.3 MakeReadWriteAtomic

▷ MakeReadWriteAtomic(*obj*) (function)

MakeReadWriteAtomic is the inverse of MakeWriteOnceAtomic (5.3.1) and MakeStrictWriteOnceAtomic (5.3.2) in that the replacement policy is being changed to being rewritable: Elements can be replaced even if they are already bound.

5.4 Thread-local records

Thread-local records allow an easy way to have a separate copy of a record for each indvidual thread that is accessed by the same name in each thread.

Example

```
gap> r := ThreadLocalRecord();; # create new thread-local record
gap> r.x := 99;;
gap> WaitThread( CreateThread( function()
>                              r.x := 100;
>                              Display(r.x);
>                              end ) );
100
gap> r.x;
99
```

As can be seen above, even though r.x is overwritten in the second thread, it does not affect the value of r.x| in the first thread

5.4.1 ThreadLocalRecord

▷ ThreadLocalRecord(*[defaults[, constructors]]*) (function)

ThreadLocalRecord creates a new thread-local record. It accepts up to two initial arguments. The *defaults* argument is a record of default values with which each thread-local copy is initially populated (this happens on demand, so values are not actually read until needed). The second argument is a record of constructors; parameterless functions that return an initial value for the respective element. Constructors are evaluated only once per thread and only if the respective element is accessed without having previously been assigned a value.

Example

```
gap> r := ThreadLocalRecord( rec(x := 99),
>      rec(y := function() return 101; end));;
gap> r.x;
99
```

```
gap> r.y;
101
gap> TaskResult(RunTask(function() return r.x; end));
99
gap> TaskResult(RunTask(function() return r.y; end));
101
```

5.4.2 SetTLDefault

▷ SetTLDefault(*record*, *name*, *value*) (function)

SetTLDefault can be used to set the default value of a record field after its creation. Here, *record* is a thread-local record, *name* is the string of the field name, and *value* is the value.

Example

```
gap> r := ThreadLocalRecord();;
gap> SetTLDefault(r, "x", 314);
gap> r.x;
314
gap> TaskResult(RunTask(function() return r.x; end));
314
```

5.4.3 SetTLConstructor

▷ SetTLConstructor(*record*, *name*, *func*) (function)

SetTLConstructor can be used to set the constructor of a thread-local record field after its creation, similar to SetTLDefault (5.4.2).

Example

```
gap> r := ThreadLocalRecord();;
gap> SetTLConstructor(r, "x", function() return 2718; end);
gap> r.x;
2718
gap> TaskResult(RunTask(function() return r.x; end));
2718
```

Chapter 6

Thread functions

HPC-GAP has low-level functionality to support explicit creation of threads. In practice, programmers should use higher-level functionality, such as tasks, to describe concurrency. The thread functions described here exist to facilitate the construction of higher level libraries and are not meant to be used directly.

6.1 Thread functions

6.1.1 CreateThread

▷ `CreateThread(`*`func[, arg1, ..., argn]`*`)` (function)

New threads are created with the function `CreateThread`. The thread takes at least one function as its argument that it will call in the newly created thread; it also accepts zero or more parameters that will be passed to that function.

The function returns a thread object describing the thread.

Only a finite number of threads can be active at a time (that limit is system-dependent). To reclaim the resources occupied by one thread, use the `WaitThread` (6.1.2) function.

6.1.2 WaitThread

▷ `WaitThread(`*`threadID`*`)` (function)

The `WaitThread` function waits for the thread identified by *`threadID`* to finish; it does not return any value. When it returns, it returns all resources occupied by the thread it waited for, such as thread-local memory and operating system structures, to the system.

6.1.3 CurrentThread

▷ `CurrentThread()` (function)

The `CurrentThread` function returns the thread object for the current thread.

6.1.4 ThreadID

▷ ThreadID(*thread*) (function)

The ThreadID function returns a numeric thread id for the given thread. The thread id of the main thread is always 0.

Example

```
gap> CurrentThread();
<thread #0: running>
gap> ThreadID(CurrentThread());
0
```

6.1.5 KillThread

▷ KillThread(*thread*) (function)

The KillThread function terminates the given thread. Any region locks that the thread currently holds will be unlocked. The thread can be specified as a thread object or via its numeric id.

The implementation for KillThread is dependent on the interpreter actually executing statements. Threads performing system calls, for example, will not be terminated until the system call returns. Similarly, long-running kernel functions will delay termination until the kernel function returns.

Use of CALL_WITH_CATCH will not prevent a thread from being terminated. If you wish to make sure that catch handlers will be visited, use InterruptThread (6.1.8) instead. KillThread should be used for threads that cannot be controlled anymore in any other way but still eat system resources.

6.1.6 PauseThread

▷ PauseThread(*thread*) (function)

The PauseThread function suspends execution for the given thread. The thread can be specified as a thread object or via its numeric id.

The implementation for PauseThread is dependent on the interpreter actually executing statements. Threads performing system calls, for example, will not pause until the system call returns. Similarly, long-running kernel functions will not pause until the kernel function returns.

While a thread is paused, the thread that initiated the pause can access the paused thread's thread-local region.

Example

```
gap> loop := function() while true do Sleep(1); od; end;;
gap> x := fail;;
gap> th := CreateThread(function() x := [1, 2, 3]; loop(); end);;
gap> PauseThread(th);
gap> x;
[ 1, 2, 3 ]
```

6.1.7 ResumeThread

▷ ResumeThread(*thread*) (function)

The ResumeThread function resumes execution for the given thread that was paused with PauseThread (6.1.6). The thread can be specified as a thread object or via its numeric id.

If the thread isn't paused, ResumeThread is a no-op.

6.1.8 InterruptThread

▷ InterruptThread(*thread, interrupt*) (function)

The InterruptThread function calls an interrupt handler for the given thread. The thread can be specified as a thread object or via its numeric id. The interrupt is specified as an integer between 0 and MAX_INTERRUPT (6.1.11).

An interrupt number of zero (or an interrupt number for which no interrupt handler has been set up with SetInterruptHandler (6.1.9) will cause the thread to enter a break loop. Otherwise, the respective interrupt handler that has been created with SetInterruptHandler (6.1.9) will be called.

The implementation for InterruptThread is dependent on the interpreter actually executing statements. Threads performing system calls, for example, will not call interrupt handlers until the system call returns. Similarly, long-running kernel functions will delay invocation of the interrupt handler until the kernel function returns.

6.1.9 SetInterruptHandler

▷ SetInterruptHandler(*interrupt, handler*) (function)

The SetInterruptHandler function allows the programmer to set up interrupt handlers for the current thread. The interrupt number must be in the range from 1 to MAX_INTERRUPT (6.1.11) (inclusive); the handler must be a parameterless function (or fail to remove a handler).

6.1.10 NewInterruptID

▷ NewInterruptID() (function)

The NewInterruptID function returns a previously unused number (starting at 1). These numbers can be used to globally coordinate interrupt numbers.

Example

```
gap> StopTaskInterrupt := NewInterruptID();
1
gap> SetInterruptHandler(StopTaskInterrupt, StopTaskHandler);
```

6.1.11 MAX_INTERRUPT

▷ MAX_INTERRUPT (global variable)

The global variable MAX_INTERRUPT (6.1.11) is an integer containing the maximum value for the interrupt arguments to InterruptThread (6.1.8) and SetInterruptHandler (6.1.9).

Chapter 7

Channels

7.1 Channels

Channels are FIFO queues that threads can use to coordinate their activities.

7.1.1 CreateChannel

▷ `CreateChannel(`*`[capacity]`*`)` (function)

`CreateChannel` returns a FIFO communication channel that can be used to exchange information between threads. Its optional argument is a capacity (positive integer). If insufficient resources are available to create a channel, it returns -1. If the capacity is not a positive integer, an error will be raised.

If a capacity is not provided, by default the channel can hold an indefinite number of objects. Otherwise, attempts to store objects in the channel beyond its capacity will block.

Example

```
gap> ch1:=CreateChannel();
<channel 0x460339c: 0 elements, 0 waiting>
gap> ch2:=CreateChannel(5);
<channel 0x460324c: 0/5 elements, 0 waiting>
```

7.1.2 SendChannel

▷ `SendChannel(`*`channel, obj`*`)` (function)

`SendChannel` accepts two arguments, a channel object returned by `CreateChannel` (7.1.1), and an arbitrary GAP object. It stores *`obj`* in *`channel`*. If *`channel`* has a finite capacity and is currently full, then `SendChannel` will block until at least one element has been removed from the channel, e.g. using `ReceiveChannel` (7.1.6).

`SendChannel` performs automatic region migration for thread-local objects. If *`obj`* is thread-local for the current thread, it will be migrated (along with all subobjects contained in the same region) to the receiving thread's thread-local data space. In between sending and receiving, *`obj`* cannot be accessed by either thread.

This example demonstrates sending messages across a channel.

Example

```
gap> ch1 := CreateChannel();;
gap> SendChannel(ch1,1);
gap> ch1;
<channel 0x460339c: 1 elements, 0 waiting>
gap> ReceiveChannel(ch1);
1
gap> ch1;
<channel 0x460339c: 0 elements, 0 waiting>
```

Sleep in the following example is used to demonstrate blocking.

Example

```
gap> ch2 := CreateChannel(5);;
gap> ch3 := CreateChannel();;
gap> for i in [1..5] do SendChannel(ch2,i); od;
gap> ch2;
<channel 0x460324c: 5/5 elements, 0 waiting>
gap> t:=CreateThread(
> function()
> local x;
> Sleep(10);
> x:=ReceiveChannel(ch2);
> Sleep(10);
> SendChannel(ch3,x);
> Print("Thread finished\n");
> end);;
> SendChannel(ch2,3); # this blocks until the thread reads from ch2
gap> ReceiveChannel(ch3); # the thread is blocked until we read from ch3
1
Thread finished
gap> WaitThread(t);
```

7.1.3 TransmitChannel

▷ TransmitChannel(*channel, obj*) (function)

TransmitChannel is identical to SendChannel (7.1.2), except that it does not perform automatic region migration of thread-local objects.

Example

```
gap> ch := CreateChannel(5);;
gap> l := [ 1, 2, 3];;
gap> original_region := RegionOf(l);;
gap> SendChannel(ch, l);
gap> WaitThread(CreateThread(function()
>      local ob; ob := ReceiveChannel(ch);
>      Display(RegionOf(ob) = original_region);
>    end));
false
gap> l := [ 1, 2, 3];;
gap> original_region := RegionOf(l);;
gap> TransmitChannel(ch, l);
```

```
gap> WaitThread(CreateThread(function()
>      local ob; ob := ReceiveChannel(ch);
>      Display(RegionOf(ob) = original_region);
>    end));
true
```

7.1.4 TrySendChannel

▷ TrySendChannel(*channel*, *obj*) (function)

TrySendChannel is identical to SendChannel (7.1.2), except that it returns if the channel is full instead of blocking. It returns true if the send was successful and false otherwise.

Example

```
gap> ch := CreateChannel(1);;
gap> TrySendChannel(ch, 99);
true
gap> TrySendChannel(ch, 99);
false
```

7.1.5 TryTransmitChannel

▷ TryTransmitChannel(*channel*, *obj*) (function)

TryTransmitChannel is identical to TrySendChannel (7.1.4), except that it does not perform automatic region migration of thread-local objects.

7.1.6 ReceiveChannel

▷ ReceiveChannel(*channel*) (function)

ReceiveChannel is used to retrieve elements from a channel. If *channel* is empty, the call will block until an element has been added to the channel via SendChannel (7.1.2) or a similar primitive.

See SendChannel (7.1.2) for an example.

7.1.7 TryReceiveChannel

▷ TryReceiveChannel(*channel*, *default*) (function)

TryReceiveChannel, like ReceiveChannel (7.1.6), attempts to retrieve an object from *channel*. If it does not succeed, however, it will return *default* rather than blocking.

Example

```
gap> ch := CreateChannel();;
gap> SendChannel(ch, 99);
gap> TryReceiveChannel(ch, fail);
99
gap> TryReceiveChannel(ch, fail);
fail
```

7.1.8 MultiSendChannel

▷ MultiSendChannel(*channel, list*) (function)

MultiSendChannel allows the sending of all the objects contained in the list *list* to *channel* as a single operation. The list must be dense and is not modified by the call. The function will send elements starting at index 1 until all elements have been sent. If a channel with finite capacity is full, then the operation will block until all elements can be sent.

The operation is designed to be more efficient than sending all elements individually via SendChannel (7.1.2) by minimizing potentially expensive concurrency operations.

See MultiReceiveChannel (7.1.10) for an example.

7.1.9 TryMultiSendChannel

▷ TryMultiSendChannel(*channel, list*) (function)

TryMultiSendChannel operates like MultiSendChannel (7.1.8), except that it returns rather than blocking if it cannot send any more elements if the channel is full. It returns the number of elements it has sent. If *channel* does not have finite capacity, TryMultiSendChannel will always send all elements in the list.

7.1.10 MultiReceiveChannel

▷ MultiReceiveChannel(*channel, amount*) (function)

MultiReceiveChannel is the receiving counterpart to MultiSendChannel (7.1.8).It will try to receive up to *amount* objects from *channel*. If the channel contains less than *amount* objects, it will return rather than blocking.

The function returns a list of all the objects received.

Example

```
gap> ch:=CreateChannel();;
gap> MultiSendChannel(ch, [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]);
gap> MultiReceiveChannel(ch,7);
[ 1, 2, 3, 4, 5, 6, 7 ]
gap> MultiReceiveChannel(ch,7);
[ 8, 9, 10 ]
gap> MultiReceiveChannel(ch,7);
[  ]
```

7.1.11 ReceiveAnyChannel

▷ ReceiveAnyChannel(*channel_1, ..., channel_n*) (function)
▷ ReceiveAnyChannel(*channel_list*) (function)

ReceiveAnyChannel is a multiplexing variant ofReceiveChannel (7.1.6). It blocks until at least one of the channels provided contains an object. It will then retrieve that object from the channel and return it.

Example

```
gap> ch1 := CreateChannel();;
gap> ch2 := CreateChannel();;
gap> SendChannel(ch2, [1, 2, 3]);;
gap> ReceiveAnyChannel(ch1, ch2);
[ 1, 2, 3 ]
```

7.1.12 ReceiveAnyChannelWithIndex

▷ `ReceiveAnyChannelWithIndex(`*channel_1, ..., channel_n*`)` (function)
▷ `ReceiveAnyChannelWithIndex(`*channel_list*`)` (function)

`ReceiveAnyChannelWithIndex` works like `ReceiveAnyChannel` (7.1.11), except that it returns a list with two elements, the first being the object being received, the second being the number of the channel from which the object has been retrieved.

Example

```
gap> ch1 := CreateChannel();;
gap> ch2 := CreateChannel();;
gap> SendChannel(ch2, [1, 2, 3]);;
gap> ReceiveAnyChannelWithIndex(ch1, ch2);
[ [ 1, 2, 3 ], 2 ]
```

7.1.13 TallyChannel

▷ `TallyChannel(`*channel*`)` (function)

`TallyChannel` returns the number of objects that a channel contains. This number can increase or decrease, as data is sent to or received from this channel. Send operations will only ever increase and receive operations will only ever decrease this count. Thus, if there is only one thread receiving data from the channel, it can use the result as a lower bound for the number of elements that will be available in the channel.

Example

```
gap> ch := CreateChannel();;
gap> SendChannel(ch, 2);
gap> SendChannel(ch, 3);
gap> SendChannel(ch, 5);
gap> TallyChannel(ch);
3
```

7.1.14 InspectChannel

▷ `InspectChannel(`*channel*`)` (function)

`InspectChannel` returns a list of the objects that a channel contains. Note that objects that are not in the shared, public, or read-only region will be temporarily stored in the so-called limbo region while in transit and will be inaccessible through normal means until they have been received.

Example

```
gap> ch := CreateChannel();;
gap> SendChannel(ch, 2);
```

```
gap> SendChannel(ch, 3);
gap> SendChannel(ch, 5);
gap> InspectChannel(ch);
[ 2, 3, 5 ]
```

This function is primarly intended for debugging purposes.

Chapter 8

Semaphores

8.1 Semaphores

Semaphores are synchronized counters; they can also be used to simulate locks.

8.1.1 CreateSemaphore

▷ `CreateSemaphore(`*`[value]`*`)` (function)

The function `CreateSemaphore` takes an optional argument, which defaults to zero. It is the counter with which the semaphore is initialized.

Example

```
gap> sem := CreateSemaphore(1);
<semaphore 0x1108e81c0: count = 1>
```

8.1.2 WaitSemaphore

▷ `WaitSemaphore(`*`sem`*`)` (function)

`WaitSemaphore` receives a previously created semaphore as its argument. If the semaphore's counter is greater than zero, it decrements the counter and returns; if the counter is zero, it waits until another thread increases it via `SignalSemaphore` (8.1.3), then decrements the counter and returns.

Example

```
gap> sem := CreateSemaphore(1);
<semaphore 0x1108e81c0: count = 1>
gap> WaitSemaphore(sem);
gap> sem;
<semaphore 0x1108e81c0: count = 0>
```

8.1.3 SignalSemaphore

▷ `SignalSemaphore(`*`sem`*`)` (function)

`SignalSemaphore` receives a previously created semaphore as its argument. It increments the semaphore's counter and returns.

Example

```
gap> sem := CreateSemaphore(1);
<semaphore 0x1108e81c0: count = 1>
gap> WaitSemaphore(sem);
gap> sem;
<semaphore 0x1108e81c0: count = 0>
gap> SignalSemaphore(sem);
gap> sem;
<semaphore 0x1108e81c0: count = 1>
```

8.1.4 Simulating locks

In order to use semaphores to simulate locks, create a semaphore with an initial value of 1. `WaitSemaphore` (8.1.2) is then equivalent to a lock operation, `SignalSemaphore` (8.1.3) is equivalent to an unlock operation.

Chapter 9

Synchronization variables

9.1 Synchronization variables

Synchronization variables (also often called dataflow variables in the literature) are variables that can be written only once; attempts to read the variable block until it has been written to.

Synchronization variables are created with `CreateSyncVar` (9.1.1), written with `SyncWrite` (9.1.2) and read with `SyncRead` (9.1.3).

Example

```
gap> sv := CreateSyncVar();;
gap> RunAsyncTask(function()
>      Sleep(10);
>      SyncWrite(sv, MakeImmutable([1, 2, 3]));
>    end);;
gap> SyncRead(sv);
[ 1, 2, 3 ]
```

9.1.1 CreateSyncVar

▷ `CreateSyncVar()` (function)

The function `CreateSyncVar` takes no arguments. It returns a new synchronization variable. There is no need to deallocate it; the garbage collector will free the memory and all related resources when it is no longer accessible.

9.1.2 SyncWrite

▷ `SyncWrite(`*`syncvar, obj`*`)` (function)

`SyncWrite` attempts to assign the value *`obj`* to *`syncvar`*. If *`syncvar`* has been previously assigned a value, the call will fail with a runtime error; otherwise, *`obj`* will be assigned to *`syncvar`*.

In order to make sure that the recipient can read the result, the *`obj`* argument should not be a thread-local object; it should be public, read-only, or shared.

9.1.3 SyncRead

▷ `SyncRead(`*`syncvar`*`)` (function)

`SyncRead` reads the value previously assigned to *`syncvar`* with `SyncWrite` (9.1.2). If no value has been assigned yet, it blocks. It returns the assigned value.

Chapter 10

Serialization support

10.1 Serialization support

HPC-GAP has support to serialize most GAP data. While functions in particular cannot be serialized, it is possible to serialize all primitive types (booleans, integers, cyclotomics, permutations, floats, etc.) as well as all lists and records.

Custom serialization support can be written for data objects, positional objects, and component objects; serialization of compressed vectors is already supported by the standard library.

10.1.1 SerializeToNativeString

▷ `SerializeToNativeString(`*obj*`)` (function)

`SerializeToNativeString` takes the object passed as an argument and turns it into a string, from which a copy of the original can be extracted using `DeserializeNativeString` (10.1.2).

10.1.2 DeserializeNativeString

▷ `DeserializeNativeString(`*str*`)` (function)

`DeserializeNativeString` reverts the serialization process.

Example:

Example

```
gap> DeserializeNativeString(SerializeToNativeString([1,2,3]));
[ 1, 2, 3 ]
```

10.1.3 InstallTypeSerializationTag

▷ `InstallTypeSerializationTag(`*type, tag*`)` (function)

`InstallTypeSerializationTag` allows the serialization of data objects, positional objects, and component objects. The value of *tag* must be unique for each type; it can be a string or integer. Non-negative integers are reserved for use by the standard library; users should use negative integers or strings instead.

Objects of such a type are serialized in a straightforward way: During serialization, data objects are converted into byte streams, positional objects into lists, and component objects into records. These objects are then serialized along with their tags; deserialization uses the type corresponding to the tag in conjunction with `Objectify` (**Reference: Objectify**) to reconstruct a copy of the original object.

Note that this functionality may be inadequate for objects that have complex data structures attached that are not meant to be replicated. The following alternative is meant for such objects.

10.1.4 InstallSerializer

▷ `InstallSerializer(`*`description, filters, method`*`)` (function)

The more general `InstallSerializer` allows for arbitarily complex serialization code. It installs *`method`* as the method to serialize objects matching *`filters`*; *`description`* has the same role as for `InstallMethod` (**Reference: InstallMethod**).

The method must return a plain list matching a specific format. The first element must be a non-negative integer, the second must be a string descriptor that is unique to the serializer; these can then be followed by an arbitrary number of arguments.

As many of the arguments (starting with the third element of the list) as specified by the first element of the list will be converted from their object representation into a serializable representation. Data objects will be converted into untyped data objects, positional objects will be converted into plain lists, and component objects into records. Conversion will not modify the objects in place, but work on copies. The remaining arguments will remain untouched.

Upon deserialization, these arguments will be passed to a function specified by the second element of the list.

Example:

Example

```
InstallSerializer("8-bit vectors", [ Is8BitVectorRep ], function(obj)
  return [1, "Vec8Bit", obj, Q_VEC8BIT(obj), IS_MUTABLE_OBJ(obj)];
end);
```

Here, `obj` will be converted into its underlying representation, while the remaining arguments are left alone. `"Vec8Bit"` is the name that is used to look up the deserializer function.

10.1.5 InstallDeserializer

▷ `InstallDeserializer(`*`descriptor, func`*`)` (function)

The *`descriptor`* value must be the same as the second element of the list returned by the serializer; *`func`* must be a function that takes as many arguments as there were arguments after the second element of that list. For deserialization, this function is invoked and needs to return the deserialized object constructed from the arguments.

Example:

Example

```
InstallDeserializer("Vec8Bit", function(obj, q, mut)
  SET_TYPE_OBJ(obj, TYPE_VEC8BIT(q, mut));
  return obj;
end);
```

Here, the untyped `obj` that was passed to the deserializer needs to be given the correct type, which is calculated from `q` and `mut`.

Chapter 11

Low-level functionality

The functionality described in this section should only be used by experts, and even by those only with caution (especially the parts that relate to the memory model).

Not only is it possible to crash or hang the GAP kernel, it can happen in ways that are very difficult to reproduce, leading to software defects that are discovered only long after deployment of a package and then become difficult to correct.

The performance benefit of using these primitives is generally minimal; while concurrency can induce some overhead, the benefit from micromanaging concurrency in an interpreted language such as GAP is likely to be small.

These low-level primitives exist primarily for the benefit of kernel programmers; it allows them to prototype new kernel functionality in GAP before implementing it in C.

11.1 Explicit lock and unlock primitives

The `LOCK` (11.1.1) operation combined with `UNLOCK` (11.1.3) is a low-level interface for the functionality of the statement.

11.1.1 LOCK

▷ `LOCK(`*`[arg_1, ..., arg_n]`*`)` (function)

`LOCK` takes zero or more pairs of parameters, where each is either an object or a boolean value. If an argument is an object, the region containing it will be locked. If an argument is the boolean value `false`, all subsequent locks will be read locks; if it is the boolean value `true`, all subsequent locks will be write locks. If the first argument is not a boolean value, all locks until the first boolean value will be write locks.

Locks are managed internally as a stack of locked regions; `LOCK` returns an integer indicating a pointer to the top of the stack; this integer is used later by the `UNLOCK` (11.1.3) operation to unlock locks on the stack up to that position. If `LOCK` should fail for some reason, it will return `fail`.

Calling `LOCK` with no parameters returns the current lock stack pointer.

11.1.2 TRYLOCK

▷ `TRYLOCK(`*`[arg_1, ..., arg_n]`*`)` (function)

TRYLOCK works similarly to LOCK (11.1.1). If it cannot acquire all region locks, it returns fail and does not lock any regions. Otherwise, its semantics are identical to LOCK (11.1.1).

11.1.3 UNLOCK

▷ UNLOCK(*stackpos*) (function)

UNLOCK unlocks all regions on the stack at *stackpos* or higher and sets the stack pointer to *stackpos*.

Example

```
gap> l1 := ShareObj([1,2,3]);;
gap> l2 := ShareObj([4,5,6]);;
gap> p := LOCK(l1);
0
gap> LOCK(l2);
1
gap> UNLOCK(p); # unlock both RegionOf(l1) and RegionOf(l2)
gap> LOCK(); # current stack pointer
0
```

11.2 Hash locks

HPC-GAP supports *hash locks*; internally, the kernel maintains a fixed size array of locks; objects are mapped to a lock via hash function. The hash function is based on the object reference, not its contents (except for short integers and finite field elements).

Example

```
gap> l := [ 1, 2, 3];;
gap> f := l -> Sum(l);;
gap> HASH_LOCK(l);   # lock 'l'
gap> f(l);           # do something with 'l'
6
gap> HASH_UNLOCK(l); # unlock 'l'
```

Hash locks should only be used for very short operations, since there is a chance that two concurrently locked objects map to the same hash value, leading to unnecessary contention.

Hash locks are unrelated to the locks used by the atomic statements and the LOCK (11.1.1) and UNLOCK (11.1.3) primitives.

11.2.1 HASH_LOCK

▷ HASH_LOCK(*obj*) (function)

HASH_LOCK (11.2.1) obtains the read-write lock for the hash value associated with obj.

11.2.2 HASH_UNLOCK

▷ HASH_UNLOCK(*obj*) (function)

HASH_UNLOCK (11.2.2) releases the read-write lock for the hash value associated with obj.

11.2.3 HASH_LOCK_SHARED

▷ HASH_LOCK_SHARED(*obj*) (function)

HASH_LOCK_SHARED (11.2.3) obtains the read-only lock for the hash value associated with obj.

11.2.4 HASH_UNLOCK_SHARED

▷ HASH_UNLOCK_SHARED(*obj*) (function)

HASH_UNLOCK_SHARED (11.2.4) releases the read-only lock for the hash value associated with obj.

11.3 Migration to the public region

HPC-GAP allows migration of arbitrary objects to the public region. This functionality is potentially dangerous; for example, if two threads try resize a plain list simultaneously, this can result in memory corruption.

Accordingly, such data should never be accessed except through operations that protect accesses through locks, memory barriers, or other mechanisms.

11.3.1 MAKE_PUBLIC

▷ MAKE_PUBLIC(*obj*) (function)

MAKE_PUBLIC (11.3.1) makes *obj* and all its subobjects members of the public region.

11.3.2 MAKE_PUBLIC_NORECURSE

▷ MAKE_PUBLIC_NORECURSE(*obj*) (function)

MAKE_PUBLIC_NORECURSE (11.3.2) makes *obj*, but not any of its subobjects members of the public region.

11.4 Memory barriers

The memory models of some processors do no guarantee that read and writes reflect accesses to main memory in the same order in which the processor performed them; for example, code may write variable v1 first, and v2 second; but the cache line containing v2 is flushed to main memory first so that other processors see the change to v2 before the change to v1.

Memory barriers can be used to prevent such counter-intuitive reordering of memory accesses.

11.4.1 ORDERED_WRITE

▷ ORDERED_WRITE(*expr*) (function)

The ORDERED_WRITE (11.4.1) function guarantees that all writes that occur prior to its execution or during the evaluation of *expr* become visible to other processors before any of the code executed after.

Example:

Example

```
gap> y:=0;; f := function() y := 1; return 2; end;;
gap> x := ORDERED_WRITE(f());
2
```

Here, the write barrier ensure that the assignment to y that occurs during the call of f() becomes visible to other processors before or at the same time as the assignment to x.

This can also be done differently, with the same semantics:

Example

```
gap> t := f();; # temporary variable
gap> ORDERED_WRITE(0);; # dummy argument
gap> x := t;
2
```

11.4.2 ORDERED_READ

▷ ORDERED_READ(*expr*) (function)

Conversely, the ORDERED_READ (11.4.2) function ensures that reads that occur before its call or during the evaluation of *expr* are not reordered with respects to memory reads occurring after it.

11.5 Object manipulation

There are two new functions to exchange a pair of objects.

11.5.1 SWITCH_OBJ

▷ SWITCH_OBJ(*obj1, obj2*) (function)

SWITCH_OBJ (11.5.1) exchanges its two arguments. All variables currently referencing *obj1* will reference *obj2* instead after the operation completes, and vice versa. Both objects stay within their previous regions.

Example

```
gap> a := [ 1, 2, 3];;
gap> b := [ 4, 5, 6];;
gap> SWITCH_OBJ(a, b);
gap> a;
[ 4, 5, 6 ]
gap> b;
[ 1, 2, 3 ]
```

The function requires exclusive access to both objects, which may necessitate using an atomic statement, e.g.:

Example

```
gap> a := ShareObj([ 1, 2, 3]);;
gap> b := ShareObj([ 4, 5, 6]);;
gap> atomic a, b do SWITCH_OBJ(a, b); od;
gap> atomic readonly a do Display(a); od;
[ 4, 5, 6 ]
gap> atomic readonly b do Display(b); od;
[ 1, 2, 3 ]
```

11.5.2 FORCE_SWITCH_OBJ

▷ FORCE_SWITCH_OBJ(*obj1, obj2*) (function)

FORCE_SWITCH_OBJ (11.5.2) works like SWITCH_OBJ (11.5.1), except that it can also exchange objects in the public region:

Example

```
gap> a := ShareObj([ 1, 2, 3]);;
gap> b := MakeImmutable([ 4, 5, 6]);;
gap> atomic a do FORCE_SWITCH_OBJ(a, b); od;
gap> a;
[ 4, 5, 6 ]
```

This function should be used with extreme caution and only with public objects for which only the current thread has a reference. Otherwise, undefined behavior and crashes can result from other threads accessing the public object concurrently.

Index

APPENDIX B. GAP MANUAL "CHANGES FROM EARLIER VERSIONS"

This Appendix contains a selection of pages from one of the four main GAP manuals from the release of GAP 4.10.2 (Month 46).

The selected pages contain the overview of GAP releases published during the project and described in Sections 3 and 4.

A current version of GAP Documentation in PDF and HTML formats could be found on the GAP website at `https://www.gap-system.org/Doc/manuals.html`.

GAP - Changes from Earlier Versions

Release 4.10.2, 19-Jun-2019

The GAP Group

The GAP Group Email: support@gap-system.org
Homepage: https://www.gap-system.org

Copyright

Contents

Chapter 1

Preface

This is one of the three main GAP books. It describes most essential changes from previous GAP releases.

In addition to this manual, there is the *GAP Tutorial* and the *GAP Reference Manual* containing detailed documentation of the mathematical functionality of GAP.

A lot of the functionality of the system and a number of contributed extensions are provided as GAP packages, and each of these has its own manual. New versions of packages are released independently of GAP releases, and changes between package versions may be described in their documentation.

Chapter 2

Changes between GAP 4.9 and GAP 4.10

This chapter contains an overview of the most important changes introduced in GAP 4.10.0 release (the first public release of GAP 4.10). Later it will also contain information about subsequent update releases for GAP 4.10.

These changes are also listed on the Wiki page https://github.com/gap-system/GAP/wiki/gap-4.10-releas

2.1 GAP 4.10.0 (November 2018)

2.1.1 New features and major changes

Reduce impact of immediate methods

GAP allows declaring so-called "immediate methods". The idea is that these are very simple and fast methods which are immediately called if information about an object becomes known, in order to perform some quick deduction. For example, if the order of a group is set, there might be immediate methods which update the filters `IsFinite` and `IsTrivial` of the group suitably.

While this can be very elegant and useful in interactive GAP sessions, the overhead for running these immediate methods and applying their results can become a major factor in the runtime of complex computations that create thousands or millions of objects.

To address this, various steps were taken:

- some immediate methods were turned into regular methods;
- a special handlers for `SetSize` was created that deduces properties which previously were taken care of by immediate methods;
- some immediate methods were replaced by implications (set via `InstallTrueMethod`), a mechanism that essentially adds zero overhead, unlike immediate methods;
- various group constructors were modified to precompute and preset properties of freshly created group objects, to avoid triggering immediate methods for these.

As a result of these and other changes, consider the following example; with GAP 4.9, it takes about 130 seconds on one test system, while with GAP 4.10 it runs in about 22 seconds, i.e., more than six times faster.

Example

```
G:=PcGroupCode( 741231213963541373679312045151639276850536621925972119311,11664);;
IsomorphismGroups(G,PcGroupCode(CodePcGroup(G),Size(G)))<>fail;
```

Relevant pull requests and issues: #2386, #2387, #2522.

Change definition of `IsPGroup` to *not* require finiteness

This is a small change in terms of amount of code changed, but we list it here as it has a potential (albeit rather unlikely) impact on the code written by GAP users: In the past, the GAP manual entry for `IsPGroup` defined p-groups as being finite groups, which differs from the most commonly used definition for p-groups. Note however that there was not actual implication installed from `IsPGroup` to `IsFinite`, so it always was possible to actually created infinite groups in the filter `IsPGroup`. In GAP 4.10, we adjusted (in #1545) the documentation for `IsPGroup` to the commonly accepted definition for p-groups. In addition, code in the GAP library and in packages using `IsPGroup` was audited and (in a very few cases) adjusted to explicitly check `IsFinite` (see e.g. #2866).

Experimental support for using the Julia garbage collector

It is now possible to use the garbage collector of the Julia language instead of GAP's traditional GASMAN garbage collector. This is partly motivated by a desire to allow tight integration with GAP and Julia in the future. Warning: right now, this is *slower*, and also requires a patched version of Julia.

Relevant pull requests: #2092, #2408, #2461, #2485, #2495, #2672, #2688, #2793, #2904, #2905, #2931.

libGAP (work in progress)

We now provide a experimental way to allow 3rd party code to link GAP as a library; this is based on the libGAP code by SageMath, but different: while we aim to provide the same functionality, we do not rename any symbols, and we do not provide the same API. We hope that a future version of SageMath can drop its custom modifications for GAP and use this interface instead. Work is underway to achieve this goal. If you are interested in this kind of interface, please get in touch with us to help us improve it. See also this email.

To get an idea how libGAP works, you can configure GAP as normal, and then execute `make testlibgap` which will build a small program that uses some of the existing API and links GAP. Relevant pull requests:

- #1690 Add a callback to `FuncJUMP_TO_CATCH`
- #2528 Add `IsLIBGAP` constant
- #2702 Add GAP kernel API
- #2723 Introduce command line options `-norepl` and `-nointeract`

2.1.2 Improved and extended functionality

- #2041 Teach `FrattiniSubgroup` methods to check for solvability
- #2053 Faster computation of modular inverses of integers
- #2057 Various changes, including:

  - Improve computation of automorphism groups for fp groups (we still recommend to instead first convert the group to a computationally nice representation, such as a perm or pc group)
  - Add `MinimalFaithfulPermutationDegree` (**Reference: MinimalFaithfulPermutationDegree**) attribute for finite groups
  - Improve performance of `GQuotients(F,G)` when `F` is an fp group
  - Some other performance and documentation tweaks

- #2061, #2086, #2159, #2306 Speed up `GcdInt`, `LcmInt`, `PValuation`, `RootInt`, `SmallestRootInt`, `IsPrimePowerInt`
- #2063 Teach GAP that BPSW pseudo primes less than 2^{64} are all known to be prime (the previous limit was 10^{13})
- #2091 Refactor `DeclareAttribute` and `NewAttribute` (arguments are now verified stricter)
- #2115, #2204, #2272 Allow (optionally) passing a random source to many more `Random` methods than before, and also to `RandomList`
- #2136 Add `shortname` entry to record returned by `IsomorphismTypeInfoFiniteSimpleGroup`
- #2181 Implement `Union(X,Y)`, where `X` and `Y` are in `PositiveIntegers`, `NonnegativeIntegers`, `Integers`, `GaussianIntegers`, `Rationals`, `GaussianRationals`, `Cyclotomics`, at least where a suitable output object exists (we already provided `Intersection(X,Y)` for a long time)
- #2185 Implement `IsCentral(M,x)`, where `M` is a magma, monoid, group, ring, algebra, etc. and `x` an element of `M` (the documentation already claimed that these exist for a long time)
- #2199 Optimize true/false conditions when coding if-statements
- #2200 Add `StringFormatted` (**Reference: StringFormatted**), `PrintFormatted` (**Reference: PrintFormatted**), `PrintToFormatted` (**Reference: PrintToFormatted**)
- #2222 Turn hidden implications into actual implications
- #2223 Add operation `PositionsBound` (**Reference: PositionsBound**) which returns the set of all bound positions in a given list
- #2224, #2243, #2340 Improve `ShowImpliedFilters` output
- #2225 Improve `LocationFunc` for kernel function
- #2232 Make `ValueGlobal` faster
- #2242 Add global function `CycleFromList` (**Reference: CycleFromList**)
- #2244 Make `rank` argument to `InstallImmediateMethod` optional, similar to `InstallMethod`
- #2274 Ensure uniform printing of machine floats `nan`, `inf`, `-inf` across different operating systems

- #2287 Turn `IsInfiniteAbelianizationGroup` into a property and add some implications involving it
- #2293, #2602, #2718 Improved and documented various kernel and memory debugging facilities (requires recompiling GAP with `-enable-debug`, `-enable-valgrind` resp. `-enable-memory-checking`)
- #2308 Method selection code was rewritten from GAP to C
- #2326 Change `SimpleGroup` to perform better input validation and improve or correct error message for type 2E
- #2375 Make `last2` and `last3` available in break loops
- #2383 Speed improvements for automorphism groups
- #2393 Track location of `InstallMethod` and `InstallImmediateMethod`
- #2422 Improve tracking of `InstallMethod` and `DeclareOperation`
- #2426 Speed up `InverseMatMod` with integer modulus
- #2427 Fix and complete support for custom functions (i.e., objects which can be called like a function using `obj(arg)` syntax)
- #2456 Add `PrintString` and `ViewString` methods for character tables
- #2474 Change `IsConstantRationalFunction` and `IsUnivariateRationalFunction` to return `false` if input isn't a rational function (instead of an error)
- #2474 Add methods for multiplying rational functions over arbitrary rings by rationals
- #2496 Finite groups whose order is known and not divisible by 4 are immediately marked as solvable
- #2509 Rewrite support for `.gz` compressed files to use zlib, now works on Windows
- #2519, #2524, #2531 `Test` now rejects empty inputs and warns if the input contains no test
- #2574 When reporting syntax errors, GAP now "underlines" the complete last token, not just the position where it stopped parsing
- #2577, #2613 Add quadratic and bilinear add forms for `Omega(e,d,q)`
- #2598 Add `BannerFunction` to `PackageInfo.g`
- #2606 Improve `PageSource` to work on functions that were read from a file given by a relative path
- #2616 Speed up computation of quotients of associative words by using existing (but previously unused) kernel functions for that
- #2640 Work on `MatrixObj` and `VectorObj`

- #2654 Make `Sortex` stable
- #2666, #2686 Add `IsBiCoset` attribute for right cosets, which is true if the right coset is also a left coset
- #2684 Add `NormalSubgroups` methods for symmetric and alternating permutation groups
- #2726 Validate `PackageInfo.g` when loading packages
- #2733 Minor performance improvements, code cleanup and very local fixes
- #2750 Reject some invalid uses of ~
- #2812 Reduce memory usage and improve performance the MTC (modified Todd-Coxeter) code that was rewritten in GAP 4.9, but which was much slower than the old (but buggy) code it replaced; the difference is now small, but the old code still is faster in some case.
- #2855, #2877 Add `IsPackageLoaded` (**Reference: IsPackageLoaded**)
- #2878 Speed up conjugacy tests for permutation by using random permutation of points when selecting base in centraliser
- #2899 `TestDirectory` reports number of failures and failed files

2.1.3 Changed documentation

- #2192 Add an example for `PRump` (**Reference: PRump**)
- #2219 Add examples to the relations chapter (see (**Reference: Relations**)).
- #2360 Document `IdealDecompositionsOfPolynomial` (**Reference: IdealDecompositionsOfPolynomial**) (also accessible via its synonym `DecomPoly`) and `NormalizerViaRadical` (**Reference: NormalizerViaRadical**)
- #2366 Do not recommend avoiding `X` which is a synonym for `Indeterminate` (**Reference: Indeterminate**)
- #2432 Correct a claim about the index of `Omega(e,p,q)` in `SO(e,p,q)` (see `SO` (**Reference: SO**))
- #2549 Update documentation of the `-T` command line option (see (**Reference: Command Line Options**))
- #2551 Add new command line option `-alwaystrace` which ensures error backtraces are printed even if break loops are disabled (see (**Reference: Command Line Options**))
- #2681 Documented `ClassPositionsOfSolvableRadical` (**Reference: ClassPositionsOfSolvableRadical**) and `CharacterTableOfNormalSubgroup` (**Reference: CharacterTableOfNormalSubgroup**)
- #2834 Improve manual section about `Info` classes (see (**Reference: Info Functions**))

2.1.4 Fixed bugs that could lead to crashes

- #2154, #2242, #2294, #2344, #2353, #2736 Fix several potential (albeit rare) crashes related to garbage collection
- #2196 Fix crash in `HasKeyBag` on SPARC Solaris 11
- #2305 Fix crash in `PartialPerm([1,2,8],[3,4,1,2]);`
- #2477 Fix crash if `~` is used to modify list
- #2499 Fix crash in the kernel functions `{8,16,32}Bits_ExponentSums3`
- #2601 Fix crash in `MakeImmutable(rec(x:=~));`
- #2665 Fix crash when an empty filename is passed
- #2711 Fix crash when tracing buggy attribute/property methods that fail to return a value
- #2766 Fix obscure crashes by using `a!{l}` syntax inside a function (this syntax never was fully implemented and was unusable, and now has been removed)

2.1.5 Fixed bugs that could lead to incorrect results

- #2085 Fix bugs in `JenningsLieAlgebra` and `PCentralLieAlgebra` that could e.g. lead to incorrect `LieLowerCentralSeries` results
- #2113 Fix `IsMonomial` for reducible characters and some related improvements
- #2183 Fix bug in `ValueMolienSeries` that could lead to `ValueMolienSeries(m,0)` not being 1
- #2198 Make multiplication of larger integers by tiny floats commutative (e.g. now $10.^{-300} * 10^{400}$ and $10^{400} * 10.^{-}300$ both give infinity, while before $10^{400} * 10.^{-300}$ gave $1.e+100$); also ensure various strange inputs, like `rec()^1;`, produce an error (instead of setting `a^1 = a` and `1*a = a` for almost any kind of object)
- #2273 Fix `TypeOfOperation` for setters of and-filters
- #2275, #2280 Fix `IsFinitelyGeneratedGroup` and `IsFinitelyGeneratedMonoid` to not (incorrectly) assume that a given infinite generating set implies that there is no finite generating set
- #2311 Do not set `IsFinitelyGeneratedGroup` for finitely generated magmas which are not groups
- #2452 Fix bug that allowed creating empty magmas in the filters `IsTrivial` and `IsMagmaWithInverses`
- #2689 Fix `LogFFE` to not return negative results on 32 bit systems
- #2766 Fix a bug that allowed creating corrupt permutations

2.1.6 Fixed bugs that could lead to break loops

- #2040 Raise error if eager float literal conversion fails (fixes #1105)
- #2582 Fix `ExtendedVectors` for trivial vector spaces
- #2617 Fix `HighestWeightModule` for Lie algebras in certain cases
- #2829 Fix `ShallowCopy` for `IteratorOfCartesianProduct`

2.1.7 Other fixed bugs

- #2220 Do not set `IsSubsetLocallyFiniteGroup` filter for finite fields
- #2268 Handle spaces in filenames of gzipped filenames
- #2269, #2660 Fix some issues with the interface between GAP and XGAP (or other similar frontends for GAP)
- #2315 Prevent creation of groups of floats, just like we prevent creation of groups of cyclotomics
- #2350 Fix prompt after line continuation
- #2365 Fix tracing of mutable variants of `One/Zero/Inv/AInv`
- #2398 Fix `PositionStream` to report correct position
- #2467 Fix support for identifiers of length 1023 and more
- #2470 Do not display garbage after certain syntax error messages
- #2533 Fix composing a map with an identity map to not produce a range that is too big
- #2638 Fix result of `Random` on 64 bit big endian system to match those on little endian, and on 32 bit big endian
- #2672 Fix `MakeImmutable` for weak pointer objects, which previously failed to make subobjects immutable
- #2674 Fix `SaveWorkspace` to return `false` in case of an error, and `true` only if successful
- #2681 Fix `Display` for the character table of a trivial group
- #2716 When seeding a Mersenne twister from a string, the last few characters would not be used if the string length was not a multiple of 4. Fixing this may lead to different series of random numbers being generated.
- #2720 Reject workspaces made in a GAP with readline support in a GAP without, and vice versa, instead of crashing
- #2657 The subobjects of the mutable values of the attributes `ComputedClassFusions`, `ComputedIndicators`, `ComputedPowerMaps`, `ComputedPrimeBlockss` are now immutable. This makes sure that the values are not accidentally changed. This change may have side-effects in users' code, for example the object returned by `0 * ComputedPowerMaps( CharacterTable( "A5" ) )[2]` had been a mutable list before the change, and is an immutable list from now on.

2.1.8 Removed or obsolete functionality

- Remove multiple undocumented internal functions. Nobody should have been using them, but if you were, you may extract it from a previous GAP release that still contained it. (#2670, #2781 and more)
- #2335 Remove several functions and variables that were deprecated for a long time: `DiagonalizeIntMatNormDriven`, `DeclarePackageDocumentation`, `KERNEL_VERSION`, `GAP_ROOT_PATHS`, `LOADED_PACKAGES`, `PACKAGES_VERSIONS`, `IsTuple`, `StateRandom`, `RestoreStateRandom`, `StatusRandom`, `FactorCosetOperation`, `ShrinkCoeffs`, `ExcludeFromAutoload`, `CharacterTableDisplayPrintLegendDefault`, `ConnectGroupAndCharacterTable`, `IsSemilatticeAsSemigroup`, `CreateCompletionFiles`, `PositionFirstComponent`, `ViewLength`
- #2502 Various kernel functions now validate their inputs more carefully (making it harder to produce bad effects by accidentally passing bad data to them)
- #2700 Forbid constructors with 0 arguments (they were never meaningful)

2.1.9 Packages

GAP 4.10.0 distribution includes 140 packages.
Added to the distribution:

- The francy package by Manuel Martins, which provides an interface to draw graphics using objects. This interface allows creation of directed and undirected graphs, trees, line charts, bar charts and scatter charts. These graphical objects are drawn inside a canvas that includes a space for menus and to display informative messages. Within the canvas it is possible to interact with the graphical objects by clicking, selecting, dragging and zooming.
- The JupyterVis package by Nathan Carter, which is intended for use in Jupyter Notebooks running GAP kernels and adds visualization tools for use in such notebooks, such as charts and graphs.

No longer redistributed with GAP:

- The linboxing package has been unusable (it does not compile) for several years now, and is unmaintained. It was therefore dropped from the GAP package distribution. If anybody is willing to take over and fix the package, the latest sources are available at `https://github.com/gap-packages/linboxing`.
- The recogbase package has been merged into the `recog` package, and therefore is no longer distributed with GAP.

2.2 GAP 4.10.1 (February 2019)

2.2.1 Changes in the core GAP system introduced in GAP 4.10.1

Fixes in the experimental way to allow 3rd party code to link GAP as a library:

- Do not start a session when loading workspace if `-nointeract` command line option is used (#2840).
- Add prototype for `GAP_Enter` and `GAP_Leave` macros (#3096).
- Prevent infinite recursions in `echoandcheck` and `SyWriteandcheck` (#3102).
- Remove `environ` arguments and `sysenviron` (#3111).

Fixes in the experimental support for using the Julia garbage collector:

- Fix task scanning for the Julia GC (#2969).
- Fix stack marking for the Julia GC (#3199).
- Specify the Julia binary instead of the Julia prefix (#3243).
- Export Julia `CFLAGS`, `LDFLAGS`, and `LIBS` to `sysinfo.gap` (#3248).
- Change `MPtr` Julia type of GAP objects to be a subtype of the abstract Julia `GapObj` type provided by the Julia package `GAPTypes.jl` (#3497).

Improved and extended functionality:

- Always generate `sysinfo.gap` (previously, it was only generated if the "compatibility mode" of the build system was enabled) (#3042).
- Add support for writing to `ERROR_OUTPUT` from kernel code (#3043).
- Add `make check` (#3285).

Changed documentation:

- Fix documentation of `NumberFFVector` (**Reference: NumberFFVector**) and add an example (#3079).

Fixed bugs that could lead to crashes:

- Fix readline crash when using autocomplete with `colored-completion-prefix` turned on in Bash (#2991).
- Fix overlapping `memcpy` in `APPEND_LIST` (#3216).

Fixed bugs that could lead to incorrect results:

- Fix bugs in the code for partial permutations (#3220).
- Fix a bug in `Gcd` for polynomials not returning standard associates, introduced in GAP 4.10.0 (#3227).

Fixed bugs that could lead to break loops:

- Change `GroupWithGenerators` (**Reference: GroupWithGenerators**) to accept collections again (to avoid regressions in code that relied on this undocumented behavior) (#3095).

- Fix `ShallowCopy` (**Reference: ShallowCopy**) for for a Knuth-Bendix rewriting system (#3128). [Reported by Ignat Soroko]
- Fix `IsMonomialMatrix` (**Reference: IsMonomialMatrix**) to work with compressed matrices (#3149). [Reported by Dominik Bernhardt]

Removed or obsolete functionality:

- Disable `make install` (previously it displayed a warning which often got ignored) (#3005).

Other fixed bugs:

- Fix some errors which stopped triggering a break loop (#3013).
- Fix compiler error with GCC 4.4.7 (#3026).
- Fix string copying logic (#3071).

2.2.2 New and updated packages since GAP 4.10.0

GAP 4.10.1 distribution contains 145 packages, including updated versions of 35 packages from GAP 4.10.0 distribution, and also the following five new packages:

- MajoranaAlgebras by Markus Pfeiffer and Madeleine Whybrow, which constructs Majorana representations of finite groups.
- PackageManager by Michael Torpey, providing a collection of functions for installing and removing GAP packages, with the eventual aim of becoming a full pip-style package manager for the GAP system.
- Thelma by Victor Bovdi and Vasyl Laver, implementing algorithms to deal with threshold elements.
- walrus by Markus Pfeiffer, providing methods for proving hyperbolicity of finitely presented groups in polynomial time.
- YangBaxter by Leandro Vendramin and Alexander Konovalov, which provides functionality to construct classical and skew braces, and also includes a database of classical and skew braces of small orders.

2.3 GAP 4.10.2 (June 2019)

2.3.1 Changes in the core GAP system introduced in GAP 4.10.21

Improvements in the experimental way to allow 3rd party code to link GAP as a library:

- Add `GAP_AssignGlobalVariable` and `GAP_IsNameOfWritableGlobalVariable` to the libGAP API (#3438).

Fixes in the experimental support for using the Julia garbage collector:

- Fix of a problem where the Julia GC during a partial sweep frees some, but not all objects of an unreachable data structure, and also may erroneously try to mark the deallocated objects (#3412).
- Fix stack scanning for the Julia GC when GAP is used as a library (#3432).

Fixed bugs that could lead to crashes:

- Fix a bug in `TransformationListList` (**Reference: TransformationListList for a source and destination**) which could cause a crash (#3463).

Fixed bugs that could lead to incorrect results:

- Fix a bug in `ClassPositionsOfLowerCentralSeries` (**Reference: ClassPositionsOfLowerCentralSeries**). [Reported by Frieder Ladisch] (#3321).
- Fix a dangerous bug in the comparison of large negative integers, introduced in GAP 4.10.1: if `x` and `y` were equal, but not identical, large negative numbers then `x < y` returned `true` instead of `false`. (#3478).

Fixed bugs that could lead to break loops:

- If the group has been obtained as subgroup from a Fitting free/solvable radical computation, the data is inherited and might not guarantee that the factor group really is Fitting free. Added a check and an assertion to catch this situation (#3154).
- Fix declaration of sparse action homomorphisms (#3281).
- `LatticeViaRadical` called `ClosureSubgroupNC` (**Reference: ClosureSubgroupNC**) assuming that the parent contained all generators. It now calls `ClosureSubgroup` (**Reference: ClosureSubgroup**) instead, since this can not be always guaranteed (this could happen, for example, in perfect subgroup computation). Also added an assertion to `ClosureSubgroupNC` (**Reference: ClosureSubgroupNC**) to catch this situation in other cases. [Reported by Serge Bouc] (#3397).
- Fix a "method not found" error in `SubdirectProduct` (**Reference: SubdirectProduct**) (#3485).

Other fixed bugs:

- Fix corner case in modified Todd-Coxeter algorithm when relator is trivial (#3311).

2.3.2 New and updated packages since GAP 4.10.1

GAP 4.10.1 distribution contains 145 packages, including updated versions of 55 packages from GAP 4.10.1 distribution,

A new package MonoidalCategories by Mohamed Barakat, Sebastian Gutsche and Sebastian Posur have been added to the distribution. It is based on the CAP package and implements monoidal structures for CAP.

Unfortunately we had to withdraw the QaoS package from distribution of GAP, as the servers it crucially relies on for its functionality have been permanently retired some time ago and are not coming back (see `https://github.com/gap-packages/qaos/issues/13` for details).

Chapter 3

Changes between GAP 4.8 and GAP 4.9

This chapter contains an overview of the most important changes introduced in GAP 4.9.1 release (the 1st public release of GAP 4.9). Later it will also contain information about subsequent update releases for GAP 4.9.

These changes are also listed on the Wiki page https://github.com/gap-system/GAP/wiki/gap-4.9-release

3.1 GAP 4.9.1 (May 2018)

3.1.1 Changes in the core GAP system introduced in GAP 4.9

Major changes:

- Merged HPC-GAP into GAP. For details, please refer to Subsection 3.1.2 at the end of these release notes.
- GAP has a new build system, which resolves many quirks and issues with the old system, and will be easier to maintain. For regular users, the usual `./configure && make` should work fine as before. If you are interested in technical details on the new build system, take a look at `README.buildsys.md`.
- The guidelines for developing GAP packages were revised and moved from the Example package to **Reference: Using and Developing GAP Packages** (#484).
- In addition to supporting single argument lambda functions like `a -> a+1`, GAP now supports lambdas with fewer or more than one argument, or even a variable number. E.g. `{a,b} -> a+b` is a shorthand for `function(a,b) return a+b; end`. For details on how to use this, see (**Reference: Function**). For technical details, e.g. why we did not choose the syntax `(a,b) -> a+b`, see #490.
- Function calls, list accesses and records accesses now can be nested. For example, you can now write `y := f().x;` (essentially equivalent to `y := f();; y := y.x;`), which previously would have resulted in an error; see #457 and #462).
- The libraries of small, primitive and transitive groups which previously were an integral part of GAP were split into three separate packages PrimgGrp, SmallGrp and TransGrp. For backwards compatibility, these are required packages in GAP 4.9 (i.e., GAP will not start without them).

We plan to change this for GAP 4.10 (see #2434), once all packages which currently implicitly rely on these new packages had time to add explicit dependencies on them (#1650, #1714).

- The performance of GAP's sorting functions (such as `Sort` (**Reference: Sort**), `SortParallel` (**Reference: SortParallel**), etc.) has been substantially improved, in some examples by more than a factor of four: as a trivial example, compare the timing for `Sort([1..100000000] * 0)`. As a side effect, the result of sorting lists with equal entries may produce different answers compared to previous GAP versions. If you would like to make your code independant of the exact employed sorting algorithm, you can use the newly added `StableSort` (**Reference: StableSort**), `StableSortBy` (**Reference: StableSortBy**) and `StableSortParallel` (**Reference: StableSortParallel**). (For some technical details, see #609).

- We removed our old home-grown big integer code, and instead always use the GMP based big integer code. This means that the GMP library now is a required dependency, not just an optional one. Note that GAP has been using GMP big integer arithmetic for a long time by default, and we also have been bundling GMP with GAP. So this change mostly removed code that was never in use for most users.

- A number of improvements have been made to `Random` (**Reference: Random**). These may lead to different sequences of numbers being created. On the up side, many more methods for `Random` (**Reference: Random**) (and other `RandomXYZ` operations) now optionally take an explicit `RandomSource` (**Reference: RandomSource**) as first argument (but not yet all: help with issue #1098 is welcome). Some relevant pull requests:

  - Allow creating random permutations using a random source (#1165)
  - Let more `Random` (**Reference: Random**) methods use an alternative source (#1168)
  - Help `Random` (**Reference: Random**) methods to use `RandomSource` (**Reference: RandomSource**) (#810)
  - Remove uses of old random generator (#808)
  - Fix `Random` (**Reference: Random**) on long (>2^28) lists (#781)
  - Fix `RandomUnimodularMat` (**Reference: RandomUnimodularMat**) (#1511)
  - Use `RandomSource` (**Reference: RandomSource**) in a few more places (#1599)

- The output and behaviour of the profiling system has been substantially improved:

  - Make profiling correctly handle the same file being opened multiple times (#1069)
  - Do not profile the `return` statements (**Reference: return**) inserted into the end of functions (#1073)
  - Ensure we reset `OutputtedFilenameList` in profiling when a workspace is restored (#1164)
  - Better tracking of amounts of memory allocated and time spent in the garbage collector (#1806)
  - Allow profiling of memory usage (#1808)
  - Remove profiling limit on files with <= 2^16 lines (#1913)

- In many cases GAP now outputs the filename and location of functions in helpful places, e.g. in error messages or when displaying compiled functions. We also try to always use the format `FILENAME:LINE`, which various utilities already know how to parse (e.g. in iTerm2, cmd-clicking on such a string can be configured to open an editor for the file at the indicated line). For some technical details, see #469, #755, #1058.
- GAP now supports constant variables, whose value cannot change anymore during runtime; code using such constants can then be slightly optimized by GAP. E.g. if `foo` is turned into a constant variable bound to the value `false`, then GAP can optimize `if foo then ... fi;` blocks completely away. For details, see `MakeConstantGlobal` (**Reference: MakeConstantGlobal**) in (**Reference: More About Global Variables**). (#1682, #1770)

Other changes:

- Enhance `StructureDescription` (**Reference: StructureDescription**) with a major rewrite, enhancing `DirectFactorsOfGroup"` and adding `SemidirectDecompositions`; the core algorithm now also works for infinite abelian groups. Further, it became faster by quickly finding abelian direct factors and recognizing several cases where the group is direct indecomposable. (#379, #763, #985)
- Mark `FittingSubgroup` (**Reference: FittingSubgroup**) and `FrattiniSubgroup` (**Reference: FrattiniSubgroup**) as nilpotent (#400)
- Add method for `Socle` (**Reference: Socle**) for finite nilpotent groups (#402)
- Change `ViewString` (**Reference: ViewString**) and `String` (**Reference: String**) methods for various inverse semigroups and monoids (#438, #880, #882)
- Enhance some nilpotent and p-group attributes (#442)
- Improve `Union` (**Reference: Union**) for a list with many ranges (#444)
- Add `UserHomeExpand` (**Reference: UserHomeExpand**), a function to expand ~ in filenames. (#447)
- Extra hint in "No Method Found" error message if one of the arguments is `fail` (#460)
- Tell Sylow subgroups of natural A_n or S_n their size when we make them (#529)
- Some small enhancements on Sylow and Hall subgroup computations, mostly for nilpotent groups. (#535)
- Remove `.zoo` archive related tools (#540)
- Add new `FrattiniSubgroup` (**Reference: FrattiniSubgroup**), `MaximalNormalSubgroups` (**Reference: MaximalNormalSubgroups**), `MinimalNormalSubgroups` (**Reference: MinimalNormalSubgroups**) and `Socle` (**Reference: Socle**) methods for abelian and/or solvable groups, even infinite ones. The new methods are only triggered if the group already knows that it is abelian and/or solvable. (#552, #583, #606)
- New attribute `NormalHallSubgroups`, returning a list of all normal Hall subgroups of a group. (#561)

- Add `ComplementClassesRepresentatives` (**Reference: ComplementClassesRepresentatives**) fallback method for arbitrary groups (#563)
- (#612) Add parsing of hex literals in strings, e.g. `"\0x61"` is turned into `"a"` (#612)
- Collection of enhancements (#683)
- Various speed improvements to polynomial factorisation and the GAP MeatAxe (#720, #1027)
- The code and documentation for transformations is improved and corrected in many instances (#727, #732)
- Change `RootFFE` to optionally takes a field or field size as first argument, from which the roots will be taken (#761)
- Change `Permanent` (**Reference: Permanent**) from a global function to an attribute (#777)
- Add `CallFuncListWrap` (**Reference: CallFuncListWrap**) to wrap return value to allow distinguishing between functions which return and functions which don't (#824)
- Allow repeated use of same `DeclareSynonym` (**Reference: DeclareSynonym**) call (#835)
- New implementation of modified Todd-Coxeter (the old one had bugs, see #302), #843)
- New functionality: Cannon/Holt automorphisms and others (#878)
- Add `IsPowerfulPGroup` (**Reference: IsPowerfulPGroup**) property, and a `FrattiniSubgroup` (**Reference: FrattiniSubgroup**) method for powerful p-groups (#894)
- Improve performance for group isomorphism/automorphisms (#896, #968)
- Make `ListX` (**Reference: ListX**), `SetX` (**Reference: SetX**), `SumX` (**Reference: SumX**) and `ProductX` (**Reference: ProductX**) support lists which are not collections (#903)
- Some improvements for `LatticeByCyclicExtension` (**Reference: LatticeByCyclicExtension**) (#905)
- Add helpers to retrieve information about operations and filters: `CategoryByName` (**Reference: CategoryByName**), `TypeOfOperation` (**Reference: TypeOfOperation**), `FilterByName` (**Reference: FilterByName**), `FiltersObj` (**Reference: FiltersObj**), `FiltersType` (**Reference: FiltersType**), `IdOfFilter`, `IdOfFilterByName`, `IsAttribute` (**Reference: IsAttribute**), `IsCategory` (**Reference: IsCategory**), `IsProperty` (**Reference: IsProperty**), `IsRepresentation` (**Reference: IsRepresentation**) (#925, #1593)
- Add case-insensitive autocomplete (#928)
- Give better error message if a help file is missing (#939)
- Add `LowercaseChar` (**Reference: LowercaseChar**) and `UppercaseChar` (**Reference: UppercaseChar**) (#952)
- Add `PositionMaximum` (**Reference: PositionMaximum**) and `PositionMinimum` (**Reference: PositionMinimum**) (#956)

- Switching default command history length from infinity to 1000 (#960)
- Allow conversion of `-infinity` to float via `NewFloat` (**Reference: NewFloat**) and `MakeFloat` (**Reference: MakeFloat**) (#961)
- Add option `NoPrecomputedData` to avoid use of data libraries in certain computations (useful if one wants to verify the content of these data libraries) (#986)
- Remove one-argument version of `AsPartialPerm` (**Reference: AsPartialPerm for a permutation and a set of positive integers**) for a transformation (#1036)
- Partial perms now have a `MultiplicativeZero` (**Reference: MultiplicativeZero**) rather than a `Zero` (**Reference: Zero**), since they are multiplicative rather than additive elements (#1040)
- Various enhancements: (#1046)
  - A bugfix in `NaturalHomomorphismByIdeal` (**Reference: NaturalHomomorphismByIdeal**) for polynomial rings
  - Improvements in handling solvable permutation groups
  - The trivial group now is a member of the perfect groups library
  - Improvements in using tabulated data for maximal subgroups
- New tests for group constructors and some fixes (e.g. `GO(1,4,5)` used to trigger an error) (#1053)
- Make `HasSolvableFactorGroup` slightly more efficient (#1062)
- Enhance `HasXXFactorGroup` (#1066)
- Remove GAP4stones from tests (#1072)
- `AsMonoid` (**Reference: AsMonoid**) and `AsSemigroup` (**Reference: AsSemigroup**) are now operations, and various bugs were resolved related to isomorphisms of semigroups and monoids (#1112)
- Mark isomorphisms between trivial groups as bijective (#1116)
- Speed up `RootMod` (**Reference: RootMod**) and `RootsMod` (**Reference: RootsMod**) for moduli with large prime factors; also add `IS_PROBAB_PRIME_INT` kernel function (#1141)
- The search for the documentation of system setters and testers now returns corresponding attributes and properties (#1144)
- Remove command line options `-c`, `-U`, `-i` and `-X`, add `-quitonbreak` (#1192, #1265, #1421, #1448)
- Remove Itanium support (#1163)
- Adding two strings now shows a more helpful error message (#1314)
- Suppress `Unbound global variable` warning in `IsBound` (**Reference: IsBound for a global variable**) (#1334)

- Increase warning level for Conway polynomial (#1363)
- Performance improvements to maximal and intermediate subgroups, fix of `RepresentativeAction` (**Reference: RepresentativeAction**) (#1390)
- Revise Chapter 52 of the reference manual (fp semigroups and monoids) (#1441)
- Improve the performance of the `Info` (**Reference: Info**) statement (#1464, #1770)
- When printing function bodies, avoid some redundant spaces (#1498)
- Add kernel functions for directly accessing entries of GF2/8bit compressed matrices (#1585)
- Add `String` (**Reference: String**) method for functions (#1591)
- Check modules were compiled with the same version of GAP when loading them (#1600)
- When printing function, reproduce `TryNextMethod()` correctly (#1613)
- New "Bitfields" feature (**Reference: Bitfields**) providing efficient support for packing multiple data items into a single word for cache and memory efficiency (#1616)
- Improved `bin/BuildPackages.sh`, in particular added option to abort upon failure (#2022)
- Rewrote integer code (GMP) for better performance of certain large integer operations, and added kernel implementations of various functions, including these:
  - Add kernel implementations of `AbsInt` (**Reference: AbsInt**), `SignInt` (**Reference: SignInt**); add new kernel functions `ABS_RAT`, `SIGN_RAT`; and speed up **Reference: mod**, `RemInt` (**Reference: RemInt**), `QuoInt` (**Reference: QuoInt**) for divisors which are small powers of 2 (#1045)
  - Add kernel implementations of `Jacobi` (**Reference: Jacobi**), `PowerModInt` (**Reference: PowerModInt**), `Valuation` (**Reference: Valuation**) (for integers), `PValuation` (**Reference: PValuation**) (for integers) (#1075)
  - Add kernel implementation of `Factorial` (**Reference: Factorial**) (#1969)
  - Add kernel implementation of `Binomial` (**Reference: Binomial**) (#1921)
  - Add kernel implementation of `LcmInt` (**Reference: LcmInt**) (#2019)
- Check version of kernel for package versions (#1600)
- Add new `AlgebraicExtensionNC` (**Reference: AlgebraicExtensionNC**) operation (#1665)
- Add `NumberColumns` and `NumberRows` to `MatrixObj` interface (#1657)
- `MinimalGeneratingSet` (**Reference: MinimalGeneratingSet**) returns an answer for non-cyclic groups that already have a generating set of size 2 (which hence is minimal) (#1755)
- Add `GetWithDefault` (**Reference: GetWithDefault**) which returns the n-th element of the list if it is bound, and the default value otherwise (#1762)
- Fast method for `ElmsBlist` when positions are a range with increment 1 (#1773)

- Make permutations remember their inverses (#1831)
- Add invariant forms for `GU(1,q)` and `SU(1,q)` (#1874)
- Implement `StandardAssociate` (**Reference: StandardAssociate**) and `StandardAssociateUnit` (**Reference: StandardAssociateUnit**) for `ZmodnZ` (**Reference: ZmodnZ**), clarify documentation for `IsEuclideanRing` (**Reference: IsEuclideanRing**) (#1990)
- Improve documentation and interface for floats (#2016)
- Add `PositionsProperty` (**Reference: PositionsProperty**) method for non-dense lists (#2021)
- Add `TrivialGroup(IsFpGroup)` (#2037)
- Change `ObjectifyWithAttributes` (**Reference: ObjectifyWithAttributes**) to return the new objects (#2098)
- Removed a never released undocumented HPC-GAP syntax extension which allowed to use a backtick/backquote as alias for `MakeImmutable` (**Reference: MakeImmutable**). (#2202).
- Various changes (#2253):
  - Improve performance and memory usage of `ImageKernelBlocksHomomorphism"`
  - Document `LowIndexSubgroups` (**Reference: LowIndexSubgroups**)
  - Correct `ClassesSolvableGroup` (**Reference: ClassesSolvableGroup**) documentation to clarify that it requires, but does not test membership
  - fix `IsNaturalGL` (**Reference: IsNaturalGL**) for trivial matrix groups with empty generating set
- Make it possible to interrupt `repeat continue; until false;` and similar tight loops with "Ctrl-C" (#2259).
- Improved GAP testing infrastructure, extended GAP test suite, and increased code coverage
- Countless other tweaks, improvements, fixes were applied to the GAP library, kernel and manual

Fixed bugs:

- Fix bugs in `NormalSubgroups` (**Reference: NormalSubgroups**) and `PrintCSV` (**Reference: PrintCSV**) (#433)
- Fix nice monomorphism dispatch for `HallSubgroup` (**Reference: HallSubgroup**) (e.g. fixes `HallSubgroup(GL(3,4), [2,3])`) (#559)
- Check for permutations whose degree would exceed the internal limit, and document that limit (#581)
- Fix segfault after quitting from the break loop in certain cases (#709 which fixes #397)

- Fix rankings for `Socle` (**Reference: Socle**) and `MinimalNormalSubgroups` (**Reference: MinimalNormalSubgroups**) (#711)
- Make key and attribute values immutable (#714)
- Make `OnTuples([-1], (1,2))` return an error (#718)
- Fix bug in `NewmanInfinityCriterion` (**Reference: NewmanInfinityCriterion**) which could corrupt the `PCentralSeries` (**Reference: PCentralSeries**) attribute (#719)
- The length of the list returned by `OnSetsPerm` is now properly set (#731)
- Fix `Remove` (**Reference: Remove**) misbehaving when last member of list with gaps in it is removed (#766)
- Fix bugs in various methods for Rees (0-)matrix semigroups: `IsFinite` (**Reference: IsFinite**), `IsOne` (**Reference: IsOne**), `Enumerator` (**Reference: Enumerator**), `IsReesMatrixSemigroup` (**Reference: IsReesMatrixSemigroup**) and `IsReesZeroMatrixSemigroup` (**Reference: IsReesZeroMatrixSemigroup**) (#768, #1676)
- Fix `IsFullTransformationSemigroup` (**Reference: IsFullTransformationSemigroup**) to work correctly for the full transformation semigroup of degree 0 (#769)
- Fix printing very large (> 2^28 points) permutations (#782)
- Fix `Intersection([])` (#854)
- Fix crash in `IsKernelFunction` for some inputs (#876)
- Fix bug in `ShortestVectors` (**Reference: ShortestVectors**) which could cause `OrthogonalEmbeddings` (**Reference: OrthogonalEmbeddings**) to enter a break loop (#941)
- Fix crash in some methods involving partial perms (#948)
- `FreeMonoid(0)` no longer satisfies `IsGroup` (**Reference: IsGroup**) (#950)
- Fix crash when invoking weak pointer functions on invalid arguments (#1009)
- Fix a bug parsing character constants (#1015)
- Fix several bugs and crashes in `Z(p,d)` for invalid arguments, e.g. `Z(4,5)`, `Z(6,3)` (#1029, #1059, #1383, #1573)
- Fix starting GAP on systems with large inodes (#1033)
- Fix `NrFixedPoints` (**Reference: NrFixedPoints for a partial perm**) and `FixedPointsOfPartialPerm` (**Reference: FixedPointsOfPartialPerm for a partial perm**) for a partial perm and a partial perm semigroup (they used to return the moved points rather than the fixed points) (#1034)
- Fix `MeetOfPartialPerms` (**Reference: MeetOfPartialPerms**) when given a collection of 1 or 0 partial perms (#1035)

- The behaviour of `AsPartialPerm` (**Reference: AsPartialPerm for a transformation and a set of positive integer**) for a transformation and a list is corrected (#1036)
- `IsomorphismReesZeroMatrixSemigroup` (**Reference: IsomorphismReesZeroMatrixSemigroup**) for a 0-simple semigroup is now defined on the zero of the source and range semigroups (#1038)
- Fix isomorphisms from finitely-presented monoids to finitely-presented semigroups, and allow isomorphisms from semigroups to fp-monoids (#1039)
- Fix `One` (**Reference: One**) for a partial permutation semigroup without generators (#1040)
- Fix `MemoryUsage` (**Reference: MemoryUsage**) for positional and component objects (#1044)
- Fix `PlainString` causing immutable strings to become mutable (#1096)
- Restore support for sparc64 (#1124)
- Fix a problem with '<' for transformations, which could give incorrect results (#1130)
- Fix crash when comparing recursive data structures such as `[~] = [~]` (#1151)
- Ensure output of `TrivialGroup(IsPermGroup)` has zero generators (#1247)
- Fix for applying the `InverseGeneralMapping` (**Reference: InverseGeneralMapping**) of an `IsomorphismFpSemigroup` (**Reference: IsomorphismFpSemigroup**) (#1259)
- Collection of improvements and fixes: (#1294)
  - A fix for quotient rings of rings by structure constants
  - Generic routine for transformation matrix to rational canonical form
  - Speed improvements to block homomorphisms
  - New routines for conjugates or subgroups with desired containment
  - Performance improvement for conjugacy classes in groups with a huge number of classes, giving significant improvements to `IntermediateSubgroups` (**Reference: IntermediateSubgroups**) (e.g. 7-Sylow subgroup in $PSL(7,2)$), ascending chain and thus in turn double coset calculations and further routines that rely on it
- Fix `EqFloat` (**Reference: EqFloat**) to return correct results, instead of always returning `false` (#1370)
- Various changes, including fixes for `CallFuncList` (**Reference: CallFuncList**) (#1417)
- Better define the result of `MappingPermListList` (**Reference: MappingPermListList**) (#1432)
- Check the arguments to `IsInjectiveListTrans` (**Reference: IsInjectiveListTrans**) to prevent crashes (#1435)
- Change `BlownUpMat` (**Reference: BlownUpMat**) to return fail for certain invalid inputs (#1488)

- Fixes for creating Green's classes of semigroups (#1492, #1771)
- Fix `DoImmutableMatrix` for finite fields (#1504)
- Make structural copy handle boolean lists properly (#1514)
- Minimal fix for algebraic extensions over finite fields of order > 256 (#1569)
- Fix for computing quotients of certain algebra modules (#1669)
- Fix an error in the default method for `PositionNot` (**Reference: PositionNot**) (#1672)
- Improvements to Rees matrix semigroups code and new tests (#1676)
- Fix `CodePcGroup` (**Reference: CodePcGroup**) for the trivial polycyclic group (#1679)
- Fix `FroidurePinExtendedAlg` for partial permutation monoids (#1697)
- Fix computing the radical of a zero dimensional associative algebra (#1701)
- Fix a bug in `RadicalOfAlgebra` (**Reference: RadicalOfAlgebra**) which could cause a break loop for some associative algebras (#1716)
- Fix a recursion depth trap error when repeatedly calling `Test` (**Reference: Test**) (#1753)
- Fix bugs in `PrimePGroup` (**Reference: PrimePGroup**) for direct products of p-groups (#1754)
- Fix `UpEnv` (**Reference: UpEnv**) (available in break loops) when at the bottom of the backtrace (#1780)
- Fix `IsomorphismPartialPermSemigroup` (**Reference: IsomorphismPartialPermSemigroup**) and `IsomorphismPartialPermMonoid` (**Reference: IsomorphismPartialPermMonoid**) for permutation groups with 0 generators (#1784)
- Fix `DisplaySemigroup` (**Reference: DisplaySemigroup**) for transformation semigroups (#1785)
- Fix "no method found" errors in `MagmaWithOne` (**Reference: MagmaWithOne**) and `MagmaWithInverses` (**Reference: MagmaWithInverses**) (#1798)
- Fix an error computing kernel of group homomorphism from fp group into permutation group (#1809)
- Fix an error in MTC losing components when copying a new augmented coset table (#1809)
- Fix output of `Where` (**Reference: Where**) in a break loop, which pointed at the wrong code line in some cases (#1814)
- Fix the interaction of signals in GAP and the IO package (#1851)
- Make line editing resilient to `LineEditKeyHandler` failure (in particular, don't crash) (#1856)
- Omit non-characters from `PermChars` (**Reference: PermChars**) results (#1867)

- Fix `ExteriorPower` (**Reference: ExteriorPowerOfAlgebraModule**) when exterior power is 0-dimensional (used to return a 1-dimensional result) (#1872)
- Fix recursion depth trap and other improvements for quotients of fp groups (#1884)
- Fix a bug in the computation of a permutation group isomorphic to a group of automorphisms (#1907)
- Fix bug in `InstallFlushableValueFromFunction` (**Reference: InstallFlushableValueFromFunction**) (#1920)
- Fix `ONanScottType` (**Reference: ONanScottType**) and introduce `RestrictedInverseGeneralMapping` (**Reference: RestrictedInverseGeneralMapping**) (#1937)
- Fix `QuotientMod` (**Reference: QuotientMod**) documentation, and the integer implementation. This partially reverts changes made in version 4.7.8 in 2013. The documentation is now correct (resp. consistent again), and several corner cases, e.g. `QuotientMod(0,0,m)` now work correctly (#1991)
- Fix `PositionProperty` (**Reference: PositionProperty**) with *from* < 1 (#2056)
- Fix inefficiency when dealing with certain algebra modules (#2058)
- Restrict capacity of plain lists to 2^28 in 32-bit and 2^60 in 64-bit builds (#2064)
- Fix crashes with very large heaps (> 2 GB) on 32 bit systems, and work around a bug in `memmove` in 32-bit glibc versions which could corrupt memory (affects most current Linux distributions) (#2166).
- Fix name of the `reversed` option in documentation of `LoadAllPackages` (**Reference: LoadAllPackages**) (#2167).
- Fix `TriangulizedMat([])` (see `TriangulizedMat` (**Reference: TriangulizedMat**) to return an empty list instead of producing an error (#2260).
- Fix several potential (albeit rare) crashes related to garbage collection (#2321, #2313, #2320).

Removed or obsolete functionality:

- Make `SetUserPreferences` obsolete (use `SetUserPreference` (**Reference: SetUserPreference**) instead) (#512)
- Remove undocumented `NameIsomorphismClass` (#597)
- Remove unused code for rational classes of permutation groups (#886)
- Remove unused and undocumented `Randomizer` and `CheapRandomizer` (#1113)
- Remove `install-tools.sh` script and documentation mentioning it (#1305)
- Withdraw `CallWithTimeout` and `CallWithTimeoutList` (#1324)
- Make `RecFields` obsolete (use `RecNames` (**Reference: RecNames**) instead) (#1331)

- Remove undocumented `SuPeRfail` and `READ_COMMAND` (#1374)
- Remove unused `oldmatint.gi` (old methods for functions that compute Hermite and Smith normal forms of integer matrices) (#1765)
- Make `TRANSDEGREES` obsolete (#1852)

3.1.2 HPC-GAP

GAP includes experimental code to support multithreaded programming in GAP, dubbed HPC-GAP (where HPC stands for "high performance computing"). GAP and HPC-GAP codebases diverged during the project, and we are currently working on unifying the codebases and incorporating the HPC-GAP code back into the mainstream GAP versions.

This is work in progress, and HPC-GAP as it is included with GAP right now still suffers from various limitations and problems, which we are actively working on to resolve. However, including it with GAP (disabled by default) considerably simplifies development of HPC-GAP. It also means that you can very easily get a (rough!) sneak peak of HPC-GAP. It comes together with the new manual book called "HPC-GAP Reference Manual" and located in the ‘doc/hpc‘ directory.

Users interested in experimenting with shared memory parallel programming in GAP can build HPC-GAP by following the instructions from https://github.com/gap-system/gap/wiki/Building-HPC-GAP. While it is possible to build HPC-GAP from a release version of GAP you downloaded from the GAP website, due to the ongoing development of HPC-GAP, we recommend that you instead build HPC-GAP from the latest development version available in the GAP repository at GitHub, i.e. https://github.com/gap-system/gap.

3.1.3 New and updated packages since GAP 4.8.10

There were 132 packages redistributed together with GAP 4.8.10. The GAP 4.9.1 distribution includes 134 packages, including numerous updates of previously redistributed packages, and some major changes outlined below.

The libraries of small, primitive and transitive groups which previously were an integral part of GAP were split into three separate packages PrimgGrp, SmallGrp and TransGrp:

- The PrimGrp package by Alexander Hulpke, Colva M. Roney-Dougal and Christopher Russell provides the library of primitive permutation groups which includes, up to permutation isomorphism (i.e., up to conjugacy in the corresponding symmetric group), all primitive permutation groups of degree < 4096.
- The SmallGrp package by Bettina Eick, Hans Ulrich Besche and Eamonn O’Brien provides the library of groups of certain “small” orders. The groups are sorted by their orders and they are listed up to isomorphism; that is, for each of the available orders a complete and irredundant list of isomorphism type representatives of groups is given.
- The TransGrp package by Alexander Hulpke provides the library of transitive groups, with an optional download of the library of transitive groups of degree 32.

For backwards compatibility, these are required packages in GAP 4.9 (i.e., GAP will not start without them). We plan to change this for GAP 4.10 (see #2434), once all packages which currently implicitly rely on these new packages had time to add explicit dependencies on them (#1650, #1714).

The new ZeroMQInterface package by Markus Pfeiffer and Reimer Behrends has been added for the redistribution. It provides both low-level bindings as well as some higher level interfaces for the ZeroMQ message passing library for GAP and HPC-GAP enabling lightweight distributed computation.

The HAPprime package by Paul Smith is no longer redistributed with GAP. Part of the code has been incorporated into the HAP package. Its source code repository, containing the code of the last distributed version, can still be found at `https://github.com/gap-packages/happrime`.

Also, the ParGAP package by Gene Cooperman is no longer redistributed with GAP because it no longer can be compiled with GAP 4.9 (see this announcement). Its source code repository, containing the code of the last distributed version, plus some first fixes needed for compatibility for GAP 4.9, can still be found at `https://github.com/gap-packages/pargap`. If somebody is interested in repairing this package and taking over its maintenance, so that it can be distributed again, please contact the GAP team.

3.2 GAP 4.9.2 (July 2018)

3.2.1 Changes in the core GAP system introduced in GAP 4.9.2

Fixed bugs that could lead to break loops:

- Fixed a bug in iterating over an empty cartesian product (#2421). [Reported by @isadofschi]

Fixed bugs that could lead to crashes:

- Fixed a crash after entering `return;` in a "method not found" break loop (#2449).
- Fixed a crash when an error occurs and `OutputLogTo` (**Reference: OutputLogTo for streams**) points to a stream which internally uses another stream (#2596).

Fixed bugs that could lead to incorrect results:

- Fixed a bug in computing maximal subgroups, which broke some other calculations, in particular, computing intermediate subgroups. (#2488). [Reported by Seyed Hassan Alavi]

Other fixed bugs and further improvements:

- Profiling now correctly handles calls to `longjmp` and allows to generate profiles using version 2.0.1 of the Profiling package (#2444).
- The `bin/gap.sh` script now respects the `GAP_DIR` environment variable (#2465). [Contributed by RussWoodroofe]
- The `bin/BuildPackages.sh` script now properly builds binaries for the simpcomp package (#2475).
- Fixed a bug in restoring a workspace, which prevented GAP from saving the history if a workspace was loaded during startup (#2578).

3.2.2 New and updated packages since GAP 4.9.1

This release contains updated versions of 22 packages from GAP 4.9.1 distribution. Additionally, it has three new packages. The new JupyterKernel package by Markus Pfeiffer provides a so-called *kernel* for the Jupyter interactive document system (https://jupyter.org/). This package requires Jupyter to be installed on your system (see https://jupyter.org/install for instructions). It also requires GAP packages IO, ZeroMQInterface, json, and also two new packages by Markus Pfeiffer called crypting and uuid, all included into GAP 4.9.2 distribution. The JupyterKernel package is not yet usable on Windows.

3.3 GAP 4.9.3 (September 2018)

3.3.1 Changes in the core GAP system introduced in GAP 4.9.3

Fixed bugs that could lead to break loops:

- Fixed a regression in `HighestWeightModule` (**Reference: HighestWeightModule**) caused by changes in sort functions introduced in GAP 4.9 release (#2617).

Other fixed bugs and further improvements:

- Fixed a compile time assertion that caused compiler error on some systems (#2691).

3.3.2 New and updated packages since GAP 4.9.2

This release contains updated versions of 18 packages from GAP 4.9.2 distribution. Additionally, it has three new packages:

- The curlInterface package by Christopher Jefferson and Michael Torpey, which provides a simple wrapper around libcurl library (https://curl.haxx.se/) to allow downloading files over http, ftp and https protocols.
- The datastructures package by Markus Pfeiffer, Max Horn, Christopher Jefferson and Steve Linton, which aims at providing standard datastructures, consolidating existing code and improving on it, in particular in view of HPC-GAP.
- The DeepThought package by Nina Wagner and Max Horn, which provides functionality for computations in finitely generated nilpotent groups given by a suitable presentation using Deep Thought polynomials.

Chapter 4

Changes between GAP 4.7 and GAP 4.8

This chapter contains an overview of the most important changes introduced in GAP 4.8.2 release (the 1st public release of GAP 4.8). Later it will also contain information about subsequent update releases for GAP 4.8. First of all, the GAP development repository is now hosted on GitHub at https://github.com/gap-system/gap, and GAP 4.8 is the first major GAP release made from this repository. The public issue tracker for the core GAP system is located at https://github.com/gap-system/gap/issues, and you may use appropriate milestones from https://github.com/gap-system/gap/milestones to see all changes that were introduced in corresponding GAP releases. An overview of the most significant ones is provided below.

4.1 GAP 4.8.2 (February 2016)

4.1.1 Changes in the core GAP system introduced in GAP 4.8

New features:

- Added support for profiling which tracks how much time in spent on each line of GAP code. This can be used to show where code is spending a long time and also check which lines of code are even executed. See the documentation for `ProfileLineByLine` (**Reference: ProfileLineByLine**) and `CoverageLineByLine` (**Reference: CoverageLineByLine**) for details on generating profiles, and the Profiling package for transforming these profiles into a human-readable form.

- Added ability to install (in the library or packages) methods for accessing lists using multiple indices and indexing into lists using indices other than positive small integers. Such methods could allow, for example, to support expressions like

Example

```
m[1,2];
m[1,2,3] := x;
IsBound(m["a","b",Z(7)]);
Unbind(m[1][2,3])
```

- Added support for partially variadic functions to allow function expressions like

Example

```
function( a, b, c, x... ) ... end;
```

which would require at least three arguments and assign the first three to `a`, `b` and `c` and then a list containing any remaining ones to `x`.

The former special meaning of the argument `arg` is still supported and is now equivalent to `function( arg... )`, so no changes in the existing code are required.

- Introduced `CallWithTimeout` and `CallWithTimeoutList` to call a function with a limit on the CPU time it can consume. This functionality may not be available on all systems and you should check `GAPInfo.TimeoutsSupported` before using this functionality. (These functions were withdrawn in GAP 4.9.)
- GAP now displays the filename and line numbers of statements in backtraces when entering the break loop.
- Introduced `TestDirectory` (**Reference: TestDirectory**) function to find (recursively) all `.tst` files from a given directory or a list of directories and run them using `Test` (**Reference: Test**).

Improved and extended functionality:

- Method tracing shows the filename and line of function during tracing.
- `TraceAllMethods` (**Reference: TraceAllMethods**) and `UntraceAllMethods` (**Reference: UntraceAllMethods**) to turn on and off tracing all methods in GAP. Also, for the uniform approach `UntraceImmediateMethods` (**Reference: UntraceImmediateMethods**) has been added as an equivalent of `TraceImmediateMethods(false)`.
- The most common cases of `AddDictionary` (**Reference: AddDictionary**) on three arguments now bypass method selection, avoiding the cost of determining homogeneity for plain lists of mutable objects.
- Improved methods for symmetric and alternating groups in the "natural" representations and removed some duplicated code.
- Package authors may optionally specify the source code repository, issue tracker and support email address for their package using new components in the `PackageInfo.g` file, which will be used to create hyperlinks from the package overview page (see `PackageInfo.g` from the Example package which you may use as a template).

Changed functionality:

- As a preparation for the future developments to support multithreading, some language extensions from the HPC-GAP project were backported to the GAP library to help to unify the codebase of both GAP 4 and HPC-GAP. The only change which is not backwards compatible is that `atomic`, `readonly` and `readwrite` are now keywords, and thus are no longer valid identifiers. So if you have any variables or functions using that name, you will have to change it in GAP 4.8.
- There was inconsistent use of the following properties of semigroups: `IsGroupAsSemigroup`, `IsMonoidAsSemigroup`, and `IsSemilatticeAsSemigroup`. `IsGroupAsSemigroup` was true for semigroups that mathematically defined a group, and for semigroups in the category `IsGroup` (**Reference: IsGroup**); `IsMonoidAsSemigroup` was only true for semigroups that

mathematically defined monoids, but did not belong to the category `IsMonoid` (**Reference: IsMonoid**); and `IsSemilatticeAsSemigroup` was simply a property of semigroups, as there is no category `IsSemilattice`.

From version 4.8 onwards, `IsSemilatticeAsSemigroup` is renamed to `IsSemilattice`, and `IsMonoidAsSemigroup` returns true for semigroups in the category `IsMonoid` (**Reference: IsMonoid**).

This way all of the properties of the type `IsXAsSemigroup` are consistent. It should be noted that the only methods installed for `IsMonoidAsSemigroup` belong to the Semigroups and Smallsemi packages.

- `ReadTest` became obsolete and for backwards compatibility is replaced by `Test` (**Reference: Test**) with the option to compare the output up to whitespaces.
- The function 'ErrorMayQuit', which differs from `Error` (**Reference: Error**) by not allowing execution to continue, has been renamed to `ErrorNoReturn` (**Reference: ErrorNoReturn**).

Fixed bugs:

- A combination of two bugs could lead to a segfault. First off, `NullMat` (**Reference: NullMat**) (and various other GAP functions), when asked to produce matrix over a small field, called `ConvertToMatrixRep` (**Reference: ConvertToMatrixRep for a list (and a field)**). After this, if the user tried to change one of the entries to a value from a larger extension field, this resulted in an error. (This is now fixed).

  Unfortunately, the C code catching this error had a bug and allowed users to type "return" to continue while ignoring the conversion error. This was a bad idea, as the C code would be in an inconsistent state at this point, subsequently leading to a crash.

  This, too, has been fixed, by not allowing the user to ignore the error by entering "return".

- The Fitting-free code and code inheriting PCGS is now using `IndicesEANormalSteps` (**Reference: IndicesEANormalSteps**) instead of `IndicesNormalSteps` (**Reference: IndicesNormalSteps**), as these indices are neither guaranteed, nor required to be maximally refined when restricting to subgroups.
- A bug that caused a break loop in the computation of the Hall subgroup for groups having a trivial Fitting subgroup.
- Including a `break` or `continue` statement in a function body but not in a loop now gives a syntax error instead of failing at run time.
- `GroupGeneralMappingByImages` (**Reference: GroupGeneralMappingByImages**) now verifies that that image of a mapping is contained in its range.
- Fixed a bug in caching the degree of transformation that could lead to a non-identity transformation accidentally changing its value to the identity transformation.
- Fixed the problem with using Windows default browser as a help viewer using `SetHelpViewer("browser");`.

4.1.2 New and updated packages since GAP 4.7.8

At the time of the release of GAP 4.7.8 there were 119 packages redistributed with GAP. New packages that have been added to the redistribution since the release of GAP 4.7.8 are:

- CAP (Categories, Algorithms, Programming) package by Sebastian Gutsche, Sebastian Posur and Øystein Skartsæterhagen, together with three associated packages GeneralizedMorphismsForCAP, LinearAlgebraForCAP and ModulePresentationsForCAP (all three - by Sebastian Gutsche and Sebastian Posur).
- Digraphs package by Jan De Beule, Julius Jonušas, James Mitchell, Michael Torpey and Wilf Wilson, which provides functionality to work with graphs, digraphs, and multidigraphs.
- FinInG package by John Bamberg, Anton Betten, Philippe Cara, Jan De Beule, Michel Lavrauw and Max Neunhöffer for computation in Finite Incidence Geometry.
- HeLP package by Andreas Bächle and Leo Margolis, which computes constraints on partial augmentations of torsion units in integral group rings using a method developed by Luthar, Passi and Hertweck. The package can be employed to verify the Zassenhaus Conjecture and the Prime Graph Question for finite groups, once their characters are known. It uses an interface to the software package 4ti2 to solve integral linear inequalities.
- matgrp package by Alexander Hulpke, which provides an interface to the solvable radical functionality for matrix groups, building on constructive recognition.
- NormalizInterface package by Sebastian Gutsche, Max Horn and Christof Söger, which provides a GAP interface to Normaliz, enabling direct access to the complete functionality of Normaliz, such as computations in affine monoids, vector configurations, lattice polytopes, and rational cones.
- profiling package by Christopher Jefferson for transforming profiles produced by `ProfileLineByLine` (**Reference: ProfileLineByLine**) and `CoverageLineByLine` (**Reference: CoverageLineByLine**) into a human-readable form.
- Utils package by Sebastian Gutsche, Stefan Kohl and Christopher Wensley, which provides a collection of utility functions gleaned from many packages.
- XModAlg package by Zekeriya Arvasi and Alper Odabas, which provides a collection of functions for computing with crossed modules and Cat1-algebras and morphisms of these structures.

4.2 GAP 4.8.3 (March 2016)

4.2.1 Changes in the core GAP system introduced in GAP 4.8.3

New features:

- New function `TestPackage` (**Reference: TestPackage**) to run standard tests (if available) for a single package in the current GAP session (also callable via `make testpackage PKGNAME=pkgname` to run package tests in the same settings that are used for testing GAP releases).

Improved and extended functionality:

- `TestDirectory` (**Reference: TestDirectory**) now prints a special status message to indicate the outcome of the test (this is convenient for automated testing). If necessary, this message may be suppressed by using the option `suppressStatusMessage`
- Improved output of tracing methods (which may be invoked, for example, with `TraceAllMethods` (**Reference: TraceAllMethods**)) by displaying filename and line number in some more cases.

Changed functionality:

- Fixed some inconsistencies in the usage of `IsGeneratorsOfSemigroup` (**Reference: IsGeneratorsOfSemigroup**).

Fixed bugs that could lead to incorrect results:

- Fallback methods for conjugacy classes, that were never intended for infinite groups, now use `IsFinite` (**Reference: IsFinite**) filter to prevent them being called for infinite groups. [Reported by Gabor Horvath]

Fixed bugs that could lead to break loops:

- Calculating stabiliser for the alternating group caused a break loop in the case when it defers to the corresponding symmetric group.
- It was not possible to use `DotFileLatticeSubgroups` (**Reference: DotFileLatticeSubgroups**) for a trivial group. [Reported by Sergio Siccha]
- A break loop while computing `AutomorphismGroup` (**Reference: AutomorphismGroup**) for `TransitiveGroup(12,269)`. [Reported by Ignat Soroko]
- A break loop while computing conjugacy classes of `PSL(6,4)`. [Reported by Martin Macaj]

Other fixed bugs:

- Fix for using Firefox as a default help viewer with `SetHelpViewer` (**Reference: SetHelpViewer**). [Reported by Tom McDonough]

4.3 GAP 4.8.4 (June 2016)

4.3.1 Changes in the core GAP system introduced in GAP 4.8.4

New features:

- The GAP distribution now includes `bin/BuildPackages.sh`, a script which can be started from the `pkg` directory via `../bin/BuildPackages.sh` and will attempt to build as many packages as possible. It replaces the `InstPackages.sh` script which was not a part of the GAP distribution and had to be downloaded separately from the GAP website. The new script is more robust and simplifies adding new packages with binaries, as it requires no adjustments if the new package supports the standard `./configure; make` build procedure.

Improved and extended functionality:

- `SimpleGroup` (**Reference: SimpleGroup**) now produces more informative error message in the case when `AtlasGroup` (**AtlasRep: AtlasGroup**) could not load the requested group.
- An info message with the suggestion to use `InfoPackageLoading` (**Reference: InfoPackageLoading**) will now be displayed when `LoadPackage` (**Reference: LoadPackage**) returns `fail` (unless GAP is started with `-b` option).
- The build system will now enable C++ support in GMP only if a working C++ compiler is detected.
- More checks were added when embedding coefficient rings or rational numbers into polynomial rings in order to forbid adding polynomials in different characteristic.

Fixed bugs that could lead to crashes:

- Fixed the crash in `-cover` mode when reading files with more than 65,536 lines.

Fixed bugs that could lead to incorrect results:

- Fixed an error in the code for partial permutations that occurred on big-endian systems. [Reported by Bill Allombert]
- Fixed the kernel method for `Remove` (**Reference: Remove**) with one argument, which failed to reduce the length of a list to the position of the last bound entry. [Reported by Peter Schauenburg]

Fixed bugs that could lead to break loops:

- Fixed the break loop while using `Factorization` (**Reference: factorization**) on permutation groups by removing some old code that relied on further caching in `Factorization`. [Reported by Grahame Erskine]
- Fixed a problem with computation of maximal subgroups in an almost simple group. [Reported by Ramon Esteban Romero]
- Added missing methods for `Intersection2` (**Reference: Intersection2**) when one of the arguments is an empty list. [Reported by Wilf Wilson]

Other fixed bugs:

- Fixed several bugs in `RandomPrimitivePolynomial` (**Reference: RandomPrimitivePolynomial**). [Reported by Nusa Zidaric]
- Fixed several problems with `Random` (**Reference: Random**) on long lists in 64-bit GAP installations.

4.4 GAP 4.8.5 (September 2016)

4.4.1 Changes in the core GAP system introduced in GAP 4.8.5

Improved and extended functionality:

- The error messages produced when an unexpected `fail` is returned were made more clear by explicitly telling that the result should not be boolean or `fail` (before it only said "not a boolean").
- For consistency, both `NrTransitiveGroups` (**transgrp: NrTransitiveGroups**) and `TransitiveGroup` (**transgrp: TransitiveGroup**) now disallow the transitive group of degree 1.

Fixed bugs that could lead to incorrect results:

- A bug in the code for algebraic field extensions over non-prime fields that may cause, for example, a list of all elements of the extension not being a duplicate-free. [Reported by Huta Gana]
- So far, `FileString` (**GAPDoc: FileString**) only wrote files of sizes less than 2G and did not indicate an error in case of larger strings. Now strings of any length can be written, and in the case of a failure the corresponding system error is shown.

Fixed bugs that could lead to break loops:

- `NaturalHomomorphismByIdeal` (**Reference: NaturalHomomorphismByIdeal**) was not reducing monomials before forming a quotient ring, causing a break loop on some inputs. [Reported by Dmytro Savchuk]
- A bug in `DefaultInfoHandler` (**Reference: DefaultInfoHandler**) caused a break loop on startup with the setting `SetUserPreference( "InfoPackageLoadingLevel", 4 )`. [Reported by Mathieu Dutour]
- The `Iterator` (**Reference: Iterator**) for permutation groups was broken when the `StabChainMutable` (**Reference: StabChainMutable for a group**) of the group was not reduced, which can reasonably happen as the result of various algorithms.

4.5 GAP 4.8.6 (November 2016)

4.5.1 Changes in the core GAP system introduced in GAP 4.8.6

Fixed bugs that could lead to break loops:

- Fixed regression in the GAP kernel code introduced in GAP 4.8.5 and breaking `StringFile` (**GAPDoc: StringFile**) ability to work with compressed files. [Reported by Bill Allombert]

4.6 GAP 4.8.7 (March 2017)

4.6.1 Changes in the core GAP system introduced in GAP 4.8.7

Fixed bugs that could lead to incorrect results:

- Fixed a regression from GAP 4.7.6 when reading compressed files after a workspace is loaded. Before the fix, if GAP is started with the `-L` option (load workspace), using `ReadLine` (**Reference: ReadLine**) on the input stream for a compressed file returned by `InputTextFile` (**Reference: InputTextFile**) only returned the first character. [Reported by Bill Allombert]

Other fixed bugs:

- Fixed compiler warning occurring when GAP is compiled with gcc 6.2.0. [Reported by Bill Allombert]

4.6.2 New and updated packages since GAP 4.8.6

This release contains updated versions of 19 packages from GAP 4.8.6 distribution. Additionally, the following package has been added for the redistribution with GAP:

- Ipres package (author: René Hartung, maintainer: Laurent Bartholdi) to work with L-presented groups, namely groups given by a finite generating set and a possibly infinite set of relations given as iterates of finitely many seed relations by a finite set of endomorphisms. The package implements nilpotent quotient, Todd-Coxeter and Reidemeister-Schreier algorithms for such groups.

4.7 GAP 4.8.8 (August 2017)

4.7.1 Changes in the core GAP system introduced in GAP 4.8.8

Fixed bugs that could lead to incorrect results:

- Fixed a bug in `RepresentativeAction` (**Reference: RepresentativeAction**) producing incorrect answers for both symmetric and alternating groups, with both `OnTuples` (**Reference: OnTuples**) and `OnSets` (**Reference: OnSets**), by producing elements outside the group. [Reported by Mun See Chang]

Fixed bugs that could lead to break loops:

- Fixed a bug in `RepresentativeAction` (**Reference: RepresentativeAction**) for S_n and A_n acting on non-standard domains.

Other fixed bugs:

- Fixed a problem with checking the path to a file when using the default browser as a help viewer on Windows. [Reported by Jack Saunders]

4.7.2 New and updated packages since GAP 4.8.7

This release contains updated versions of 29 packages from GAP 4.8.7 distribution. Additionally, the Gpd package (author: Chris Wensley) has been renamed to Groupoids.

APPENDIX C. SHARING REPRODUCIBLE COMPUTATIONAL EXPERIMENTS

This appendix shows how GAP code and data may be organized in a reproducible computational experiment which can be run on a freely available service called Binder. It demonstrates integration of a number of concepts mentioned in the report: GAP regression testing (Subsection 5.1); GAP Docker containers (Subsection 5.2); setup for continuous integration and code coverage reports for GAP packages (Subsection 5.4); and the GAP Jupyter interface (Subsection 3.5).

For this demonstrator, we use the supplementary code for the paper [5] by Alexandre Borovik and Şükrü Yalçınkaya. In this paper they present a polynomial time algorithm for solving a major problem in computational group theory, which remained open since 1999 [2]. The code implementing this algorithm is located in the file `unipoly.g` at `https://github.com/sukru-yalcinkaya/unipoly`.

Presently, the authors do not intent to organise their code in the form of a new GAP package; nevertheless it can reuse packages setup for Travis CI and Codecov by creating a `tst` directory with the test files and adapting `.travis.yml` and `.codecov.yml` configuration files from the GAP package EXAMPLE. Connecting their repository to Travis CI and Codecov, the authors will be able to automatically check that the code works in GAP 4.9, GAP 4.10 and the master branch of the GAP repository, which is a prototype for GAP 4.11.

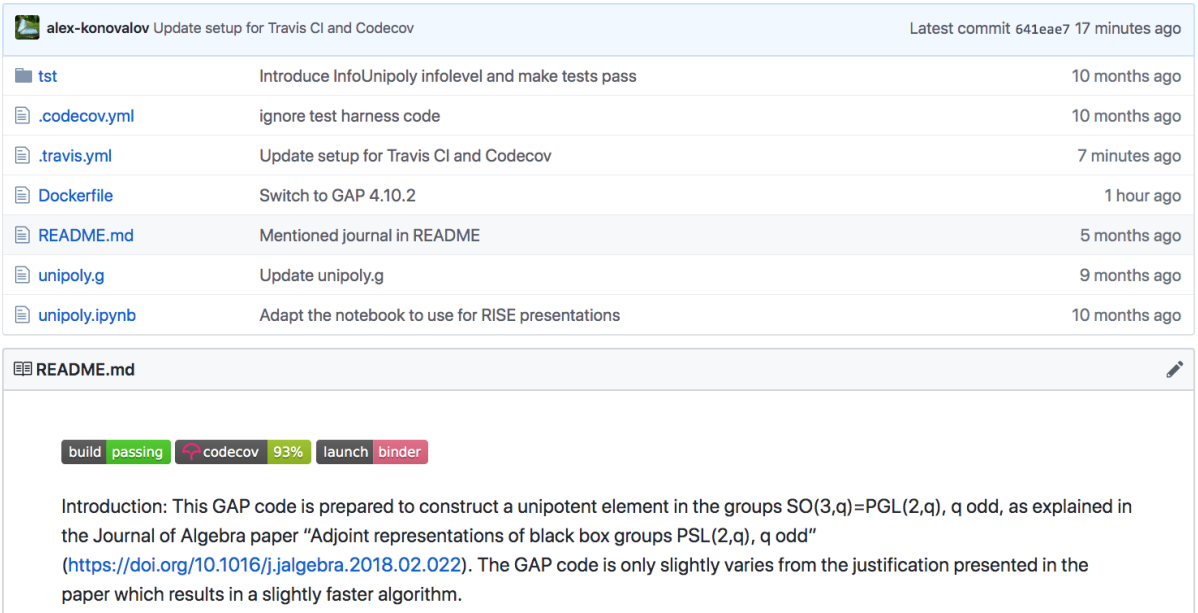


FIGURE 18. unipoly project repository

The next step is to produce a Jupyter notebook which combines input, output, and textual narrative in one document. The notebook contains the `Read(unipoly.g);` command to read the code first. This is a good practice for organizing reproducible experiments: keeping the code in a single location in a `.g` file allows its reuse and automated testing, and avoids code duplication.

When the authors prepared and committed the Jupyter notebook describing their calculation, they are ready to share it on Binder. First they need to add to their repository a `Dockefile` with content displayed on Figure 19. This file contains instructions for building a new Docker container based on the container `gapsystem/gap-docker` with the latest GAP release (see Subsection 5.2). Additional commands specify how to copy the code into the new container, and install additional extensions for Jupyter-based slideshows.

```
FROM gapsystem/gap-docker

MAINTAINER Alexander Konovalov <alexander.konovalov@st-andrews.ac.uk>

COPY --chown=1000:1000 . $HOME/unipoly

RUN sudo pip3 install ipywidgets RISE

RUN jupyter-nbextension install rise --user --py

RUN jupyter-nbextension enable rise --user --py

USER gap

WORKDIR $HOME/unipoly
```

FIGURE 19. Docker file for Binder.

After that one should follow Binder instructions to set up a new Binder project. A completed setup will allow to click on the "launch binder" button in the README file on GitHub (as seen on Figure 18 to start a Jupyter notebook server in the cloud, either using a prebuilt image of the project or by building a new one in case of any changes in the GitHub repository. When the server will be started, it will display a file browser as shown on Figure 20.

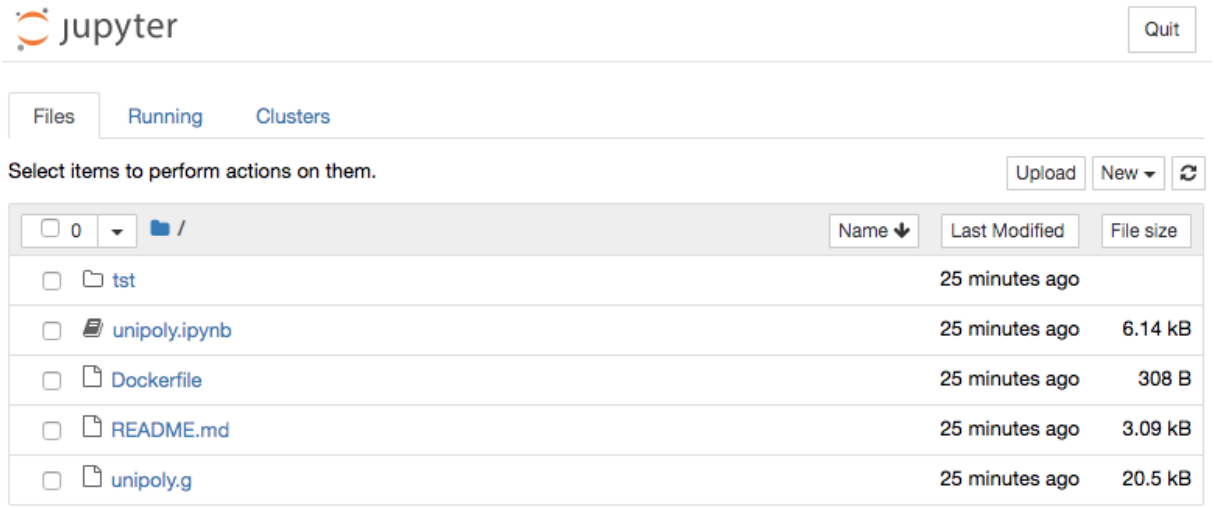


FIGURE 20. File browser running on Binder

Clicking on `unipoly.ipynb`, the user will open the Jupyter notebook and will be able to rerun it as shown on Figure 21.

One could also switch to the slideshow mode (the notebook would require some configuration to explain which cells should be displayed on the same slide and which not), and run an interactive presentation, as as shown on Figure 22.

Thus, connecting the project repository to Binder allows other users (e.g. readers or referees of the paper) to rerun computations on Binder instead of on installing and configuring all necessary components themselves, which may not only require additional time and expertise, but also be not possible due to missing dependencies, incompatible operating system, non-administrative permissions etc. On the other hand an expert will still be able to clone the repository and reproduce the experiment running the same Jupyter notebook locally, without runtime and memory limits imposed by Binder.

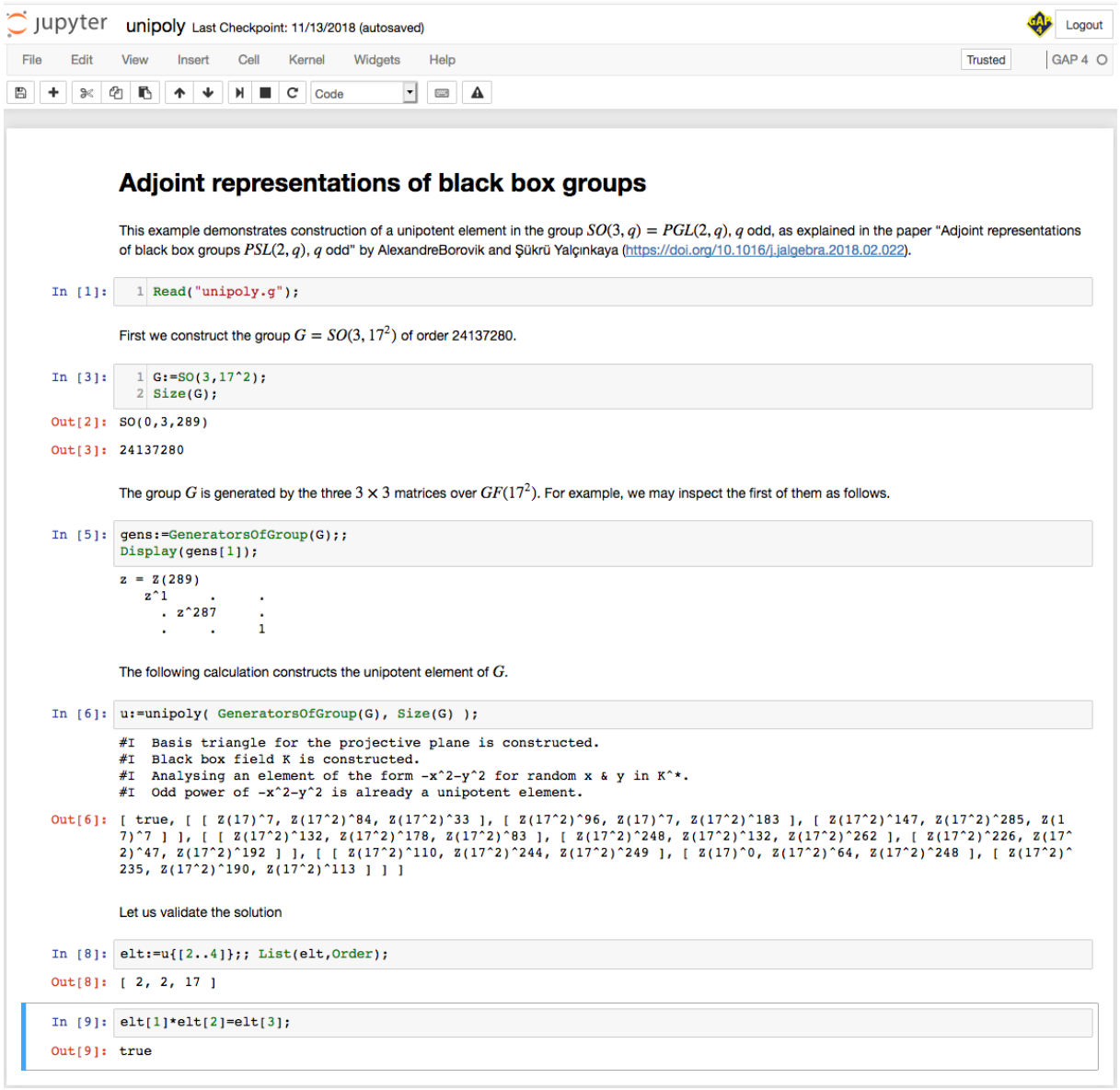


FIGURE 21. Jupyter notebook running on Binder

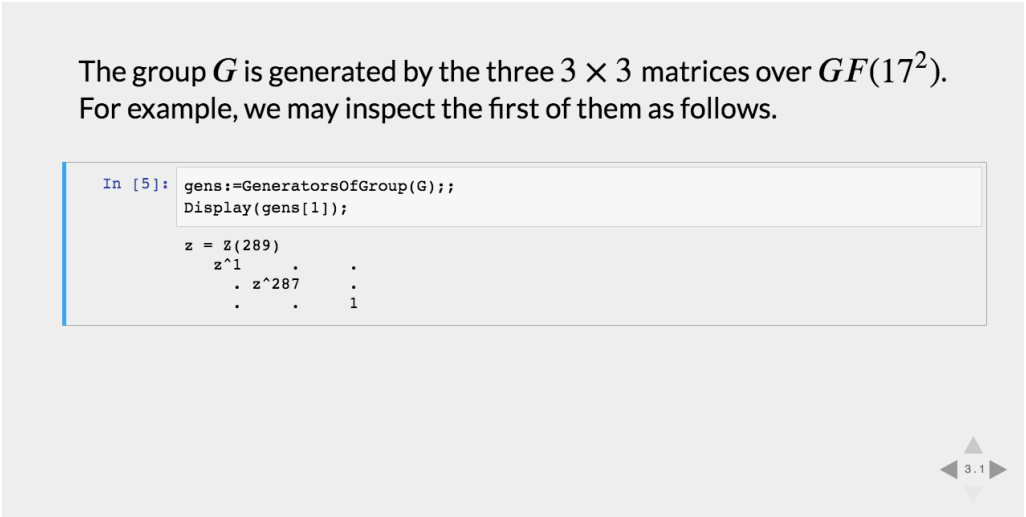


FIGURE 22. Slide with interactive computation running on Binder

APPENDIX D. LIBSEMIGROUPS IN GAP

This Appendix shows an example of LIBSEMIGROUPS being run from inside GAP via the SEMIGROUPS package, as discussed in Section 3.4. The full interactive notebook can be accessed at `https://mybinder.org/v2/gh/OpenDreamKit/gap-demos/master?filepath=Semigroups.ipynb`.

Semigroups package in GAP

```
In [1]: LoadPackage("Semigroups");
        #I  method installed for Matrix matches more than one declaration
Out[1]: true
```

The *Semigroups* package for GAP contains an external library called *libsemigroups*.

This library is called automatically from GAP whenever the appropriate questions are asked. Let's see the library in action.

We start by creating the *Motzkin Monoid* of degree 4. This is a bipartition semigroup, which has recently been of interest to researchers.

```
In [3]: M := MotzkinMonoid(4);
        Size(M);
Out[2]: <regular bipartition *-monoid of size 323, degree 4 with 8 generators>
Out[3]: 323
```

The Semigroups manual (http://gap-packages.github.io/Semigroups/doc/chap6.html) tells us that the *Froidure--Pin* algorithm can be used with semigroups that are *enumerable*. Furthermore, it tells us that *libsemigroups* is used if the semigroup in question is a semigroups of bipartitions. We first check that this is true:

```
In [5]: IsEnumerableSemigroupRep(M);
        IsBipartitionSemigroup(M);
Out[4]: true
Out[5]: true
```

So, if we ask the appropriate questions, *libsemigroups* itself will be called. For example, it will be used when we ask for the semigroup's $\mathcal{D}$-classes:

```
In [7]: d := GreensDClasses(M);
        Length(d);
Out[6]: [ <Green's D-class: <block bijection: [ 1, -1 ], [ 2, -2 ], [ 3, -3 ], [ 4, -4 ]>>, <Green's D-class: <bipartitio
        n: [ 1, 2 ], [ 3, -3 ], [ 4, -4 ], [ -1, -2 ]>>, <Green's D-class: <bipartition: [ 1 ], [ 2, -1 ], [ 3, -2 ], [ 4,
        -3 ], [ -4 ]>>, <Green's D-class: <bipartition: [ 1, 2 ], [ 3, 4 ], [ -1, -2 ], [ -3, -4 ]>>, <Green's D-class: <b
        ipartition: [ 1, 2 ], [ 3, -1 ], [ 4 ], [ -2, -3 ], [ -4 ]>> ]
Out[7]: 5
```

We know that the Motzkin monoid has one $\mathcal{D}$-class for each rank, 0 to 4.

```
In [8]: List(d, class -> RankOfBipartition(Representative(class)));
Out[8]: [ 4, 2, 3, 0, 1 ]
```

Let's take elements of rank 1 and 2, and use them to generate a congruence.

```
In [10]: x := Random(d[5]);
         y := Random(d[2]);
Out[9]: <bipartition: [ 1 ], [ 2, -4 ], [ 3, 4 ], [ -1 ], [ -2 ], [ -3 ]>
Out[10]: <bipartition: [ 1, -1 ], [ 2 ], [ 3, -2 ], [ 4 ], [ -3, -4 ]>
```

```
In [12]: RankOfBipartition(x);
         RankOfBipartition(y);
Out[11]: 1
Out[12]: 2
```

```
In [13]: cong := SemigroupCongruence(M, [x, y]);;
```

Calculating features of the congruence will also use the *libsemigroups* library, even taking advantage of parallelism to try different algorithms at the same time!

```
In [14]: NrCongruenceClasses(cong);
Out[14]: 18
```

```
In [15]: IsReesCongruence(cong);
Out[15]: true
```

All these features are faster for using the library, and this has even made it possible to calculate entire congruence lattices in a reasonable amount of time.

```
In [16]: latt := LatticeOfCongruences(M);
Out[16]: <digraph with 11 vertices, 57 edges>
```

```
In [17]: time;
Out[17]: 20895
```

The lattice has an interesting structure, but it is not immediately obvious why it has the shape it does. This investigation led to a classification of the congruences of a variety of related monoids in the following research paper:

J. East, J. D. Mitchell, N. Ruškuc, and M. Torpey, *Congruence lattices of finite diagram monoids*, Advances in Mathematics, **333**:931–1003, 2018, https://doi.org/10.1016/j.aim.2018.05.016.

REFERENCES

[1] Martin R. Albrecht. "The M4RIE library for dense linear algebra over small fields with even characteristic". In: *CoRR* abs/1111.6900 (2011). arXiv: 1111.6900. URL: http://arxiv.org/abs/1111.6900.

[2] L Babai and R Beals. "A polynomial-time theory of black box groups I". In: *Groups St Andrews 1997 in Bath*. Ed. by C. M. Campbell et al. Vol. 1. London Mathematical Society Lecture Note Series. Cambridge University Press, 1999, pp. 30–64. DOI: 10.1017/CBO9781107360228.004.

[3] J. Bamberg et al. *FinInG – Finite Incidence Geometry, Version 1.4.1*. http://www.fining.org. Refereed GAP package. 2018.

[4] Reimer Behrends et al. "HPC-GAP: engineering a 21st-century high-performance computer algebra system". In: *Concurrency and Computation: Practice and Experience* 28.13 (2016), pp. 3606–3636. DOI: 10.1002/cpe.3746. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.3746. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.3746.

[5] A. Borovik and Ş. Yalçınkaya. "Adjoint representations of black box groups $\mathrm{PSL}_2(\mathbb{F}_q)$". In: *J. Algebra* 506 (2018), pp. 540–591. ISSN: 0021-8693. DOI: 10.1016/j.jalgebra.2018.02.022. URL: https://doi.org/10.1016/j.jalgebra.2018.02.022.

[6] T. Breuer and S. Linton. "The GAP 4 Type System. Organizing Algebraic Algorithms". In: *ISSAC '98: Proceedings of the 1998 international symposium on Symbolic and algebraic computation*. Chairman: Volker Weispfenning and Barry Trager. ACM. Rostock, Germany: ACM Press, 1998, 38–45. ISBN: 1-58113-002-3.

[7] N. Carter. *JupyterViz – Visualization Tools for Jupyter and the GAP REPL, Version 1.5.1*. https://nathancarter.github.io/jupyterviz. GAP package. 2019.

[8] N. Carter. *SemigroupViz – Visualization Tools for Semigroups, Version 1.0.0*. https://nathancarter.github.io/semigroupviz. GAP package. 2019.

[9] Nathan Carter and Ray Ellis. *Group Explorer – Visualization software for the abstract algebra classroom*. 2019. URL: https://nathancarter.github.io/group-explorer/.

[10] Nathan C. Carter. *Visual group theory*. Classroom Resource Materials Series. Mathematical Association of America, Washington, DC, 2009, pp. xiv+297. ISBN: 978-0-88385-757-1.

[11] Gene Cooperman. *Parallel GAP/MPI (ParGAP/MPI), Version 1*. http://www.ccs.neu.edu/home/gene/pargap.html. 1999.

[12] J. De Beule et al. *Digraphs – Graphs, digraphs, and multidigraphs in GAP, Version 0.15.2*. https://gap-packages.github.io/Digraphs. GAP package. 2019.

[13] S. Gutsche and M. Horn. *AutoDoc – Generate documentation from GAP source code, Version 2019.05.20*. https://gap-packages.github.io/AutoDoc. GAP package. 2019.

[14] S. Gutsche, M. Horn, and C. Söger. *NormalizInterface – GAP wrapper for Normaliz, Version 1.1.0*. https://gap-packages.github.io/NormalizInterface. GAP package. 2019.

[15] A. Hulpke. *matgrp – Matric Group Interface Routines, Version 0.62*. http://www.math.colostate.edu/~hulpke/matgrp. GAP package. 2019.

[16] C. Jefferson. *ferret – Backtrack Search in Permutation Groups for GAP, Version 1.0.2*. https://gap-packages.github.io/ferret/. GAP package. 2019.

[17] C. Jefferson. *profiling – Line by line profiling and code coverage for GAP, Version 2.2.1*. https://gap-packages.github.io/profiling/. GAP package. 2019.

[18] C. Jefferson and M. Torpey. *curlInterface – Simple Web Access, Version 2.1.1*. https://gap-packages.github.io/curlInterface/. GAP package. 2018.
[19] A. Konovalov and S. Linton. *SCSCP – Symbolic Computation Software Composability Protocol in GAP, Version 2.3.0*. https://gap-packages.github.io/scscp. Refereed GAP package. 2019.
[20] S. Linton, R. Parker, and M. Pfeiffer. *meataxe64 – low-level GAP bindings to meataxe64, Version 0.1*. https://gap-packages.github.io/meataxe64/. GAP package. 2019.
[21] Nicholas James Loughlin. "Understanding idempotents in diagram semigroups". PhD thesis. Newcastle University, 2018.
[22] M. Martins. *Francy – Framework for Interactive Discrete Mathematics, Version 1.2.4*. https://gap-packages.github.io/francy. GAP package. 2019.
[23] J. Mitchell. *Semigroups – A package for semigroups and monoids, Version 3.1.3*. https://gap-packages.github.io/Semigroups. GAP package. 2019.
[24] J. D. Mitchell, M. Torpey, et al. *libsemigroups – Library for semigroups and monoids, Version 0.6.7*. https://libsemigroups.github.io/libsemigroups/. C++ library. 2019.
[25] W. Nickel, G. Gamble, and A. Konovalov. *Example – Example/Template of a GAP Package, Version 4.1.1*. https://gap-packages.github.io/example. GAP package. 2018.
[26] M. Pfeiffer. *crypting – Hashes and Crypto in GAP, Version 0.9*. https://gap-packages.github.io/crypting/. GAP package. 2018.
[27] M. Pfeiffer. *uuid – RFC 4122 UUIDs, Version 0.6*. https://gap-packages.github.io/uuid/. GAP package. 2018.
[28] M. Pfeiffer. *walrus – A new approach to proving hyperbolicity, Version 0.99*. https://gap-packages.github.io/walrus. GAP package. 2019.
[29] M. Pfeiffer, R. Behrends, and the GAP Team. *ZeroMQInterface – ZeroMQ bindings for GAP, Version 0.11*. https://gap-packages.github.io/ZeroMQInterface/. GAP package. 2018.
[30] M. Pfeiffer, M. Martins, and the GAP Team. *JupyterKernel – Jupyter kernel written in GAP, Version 1.3*. https://gap-packages.github.io/JupyterKernel/. GAP package. 2019.
[31] M. Pfeiffer and M. Whybrow. *MajoranaAlgebras – A package for constructing Majorana algebras and representations, Version 1.4*. https://MWhybrow92.github.io/MajoranaAlgebras/. GAP package. 2018.
[32] M. Pfeiffer et al. *datastructures – Collection of standard data structures for GAP, Version 0.2.3*. https://gap-packages.github.io/datastructures. GAP package. 2018.
[33] M. Torpey. *PackageManager – Easily download and install GAP packages, Version 0.4*. https://gap-packages.github.io/PackageManager/. GAP package. 2019.
[34] L. Vendramin and A. Konovalov. *YangBaxter – Combinatorial Solutions for the Yang-Baxter equation, Version 0.8.0*. https://gap-packages.github.io/YangBaxter. GAP package. 2019.

Disclaimer: this report, together with its annexes and the reports for the earlier deliverables, is self contained for auditing and reviewing purposes. Hyperlinks to external resources are meant as a convenience for casual readers wishing to follow our progress; such links have been checked for correctness at the time of submission of the deliverable, but there is no guarantee implied that they will remain valid.