libGAP

GAP kernel as a C shared library

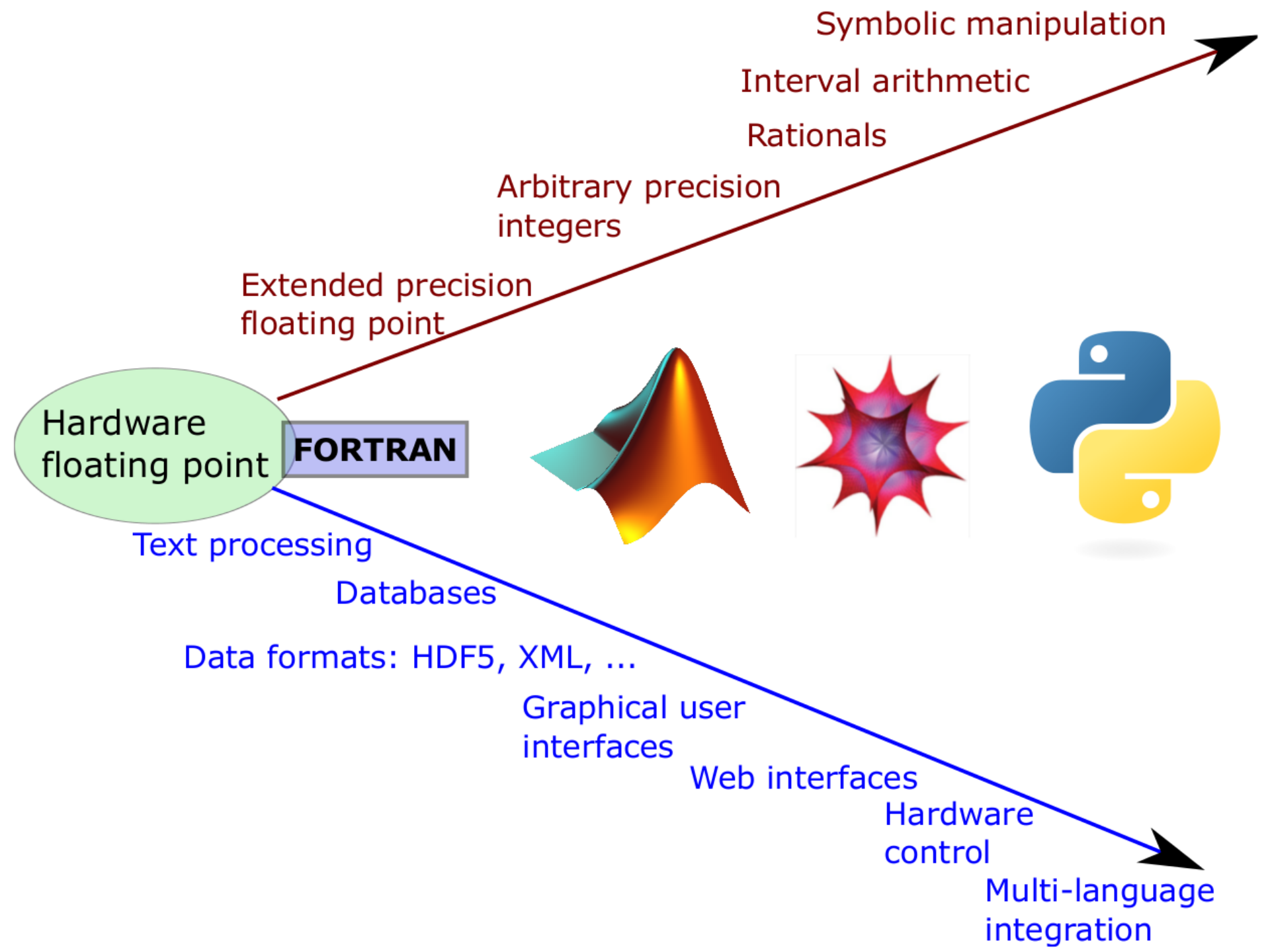
Motivation

Toric geometry

- 473800776 reflexive 4-d polytopes
- Each defines a toric variety
- Crosses different mathematical specialities: Convex geometry, algebraic geometry, groups, graphs, linear algebra, ...

Need low latency, can't use pexpect

External code must have a C shared library interface


Symbolic manipulation
Interval arithmetic
Rationals
Arbitrary precision integers
Extended precision floating point
Hardware floating point
FORTRAN
Text processing
Databases
Data formats: HDF5, XML, ...
Graphical user interfaces
Web interfaces
Hardware control
Multi-language integration

libGAP

- Small kernel ideally suited for a library
- KISS (keep it simple super): don't invent a library interface
- Just export the GAP kernel symbols

Goal:

1. Evaluate string, get output string
2. Access GAP variables on the C level

Build System

No sane way to build shared libraries without libtool

Step 1: Take the GAP src folder and wrap it in a standard autotools build system.

```
[vbraun@laptop libgap]$ ./configure --help
`configure' configures libGAP 4.6.4 to adapt to many kinds of systems.
Optional Packages:
  --with-gmp=<path>       prefix of GMP installation.
  --with-sage=<path>      the SAGE_LOCAL path
```

libGAP usage

- libgap_initialize passes arguments to InitializeGap
- Pass input to GAP: SyFgets
- Read output to GAP: PutChrTo
- Or call any GAP kernel function!

Naming Conventions

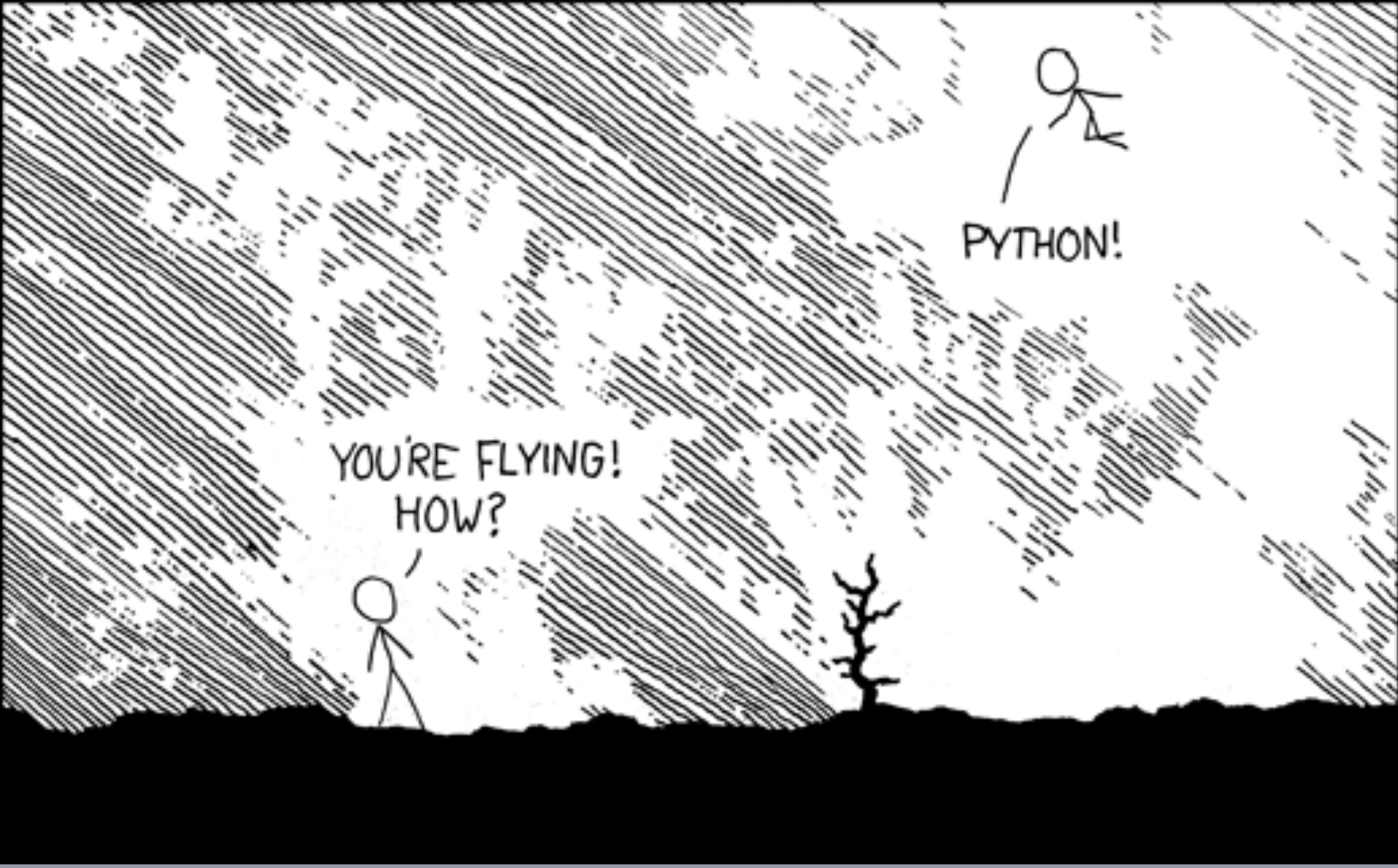
- All GAP symbols (C-level names) are prefixed with libGAP_<...> to avoid collisions
- Upstreams sources are transformed with Python PLY (parse/lex/yacc)
- Symbols that I added start with libgap_<...>

Garbage Collector

GASMAN (the GAP garbage collector) is:

- compacting (moves memory bags)
- inspects the stack: memory bags referenced by stack-allocated variables are kept alive
- Need to tell LibGAP where the C stack is
- Wrap all calls into GAP functions in libgap_enter() / libgap_exit() blocks.

```
void eval(char* input)
{
  libgap_start_interaction(input);
  libGAP_ExecStatus status;
  libgap_enter();
  status = libGAP_ReadEvalCommand
                  (libGAP_BottomLVars);
  /* error handling skipped */
  libGAP_ViewObjHandler
                    (libGAP_ReadEvalResult);
  libgap_exit();
  char* out = libgap_get_output();
  printf("Output follows...\n%s", out);
  libgap_finish_interaction();
}
```



Using libGAP from Sage/Python

Interaction with GAP

Can also interact with the GAP kernel by calling it from C:

```
libGAP_Obj obj =
    libGAP_INTOBJ_INT(123);
```

For Sage, we want a Cython wrapper of GAP objects

Cython is one of the key Sage technologies, a language extension for Python that allows using C from Python without the boilerplate code.

```
def command(command_string):
    libgap_start_interaction(command_string)
    libgap_enter()
    status = libGAP_ReadEvalCommand(
                    libGAP_BottomLVars)
    libGAP_ViewObjHandler(
                    libGAP_ReadEvalResult)
    libgap_exit()
    s = libgap_get_output()
    print 'Output follows...'
    print s
    libgap_finish_interaction()
```

GAP object wrapper

```
cdef class GapElement(RingElement):
    # the pointer to the GAP object
    cdef libGAP_Obj value
    cdef _initialize(self, parent,
                     libGAP_Obj obj)
    cpdef is_bool(self)
```

Construct from any suitable Sage input:

```
sage: a = libgap(123)
sage: a^2
15129
```

```
sage: lst = libgap('[]')
sage: lst.Add(1)
sage: lst.Add(5)
sage: lst.Add(7)
sage: lst
[ 1, 5, 7 ]
sage: len(lst)
3
sage: lst[0]
1
sage: [ x^2 for x in lst ]
[1, 25, 49]
sage: type(_[0])
<type 'sage.libs.gap.libgap.GapElement'>
```

Some examples of syntactic sugar

More complicated types

GAP record = Python dictionary

```
sage: rec = libgap({'foo':'bar'});  rec
rec( foo := "bar" )
sage: rec['foo']
"bar"
```

Finite Fields

```
sage: F.<a> = GF(4)
sage: libgap(1+a)
Z(2^2)^2
sage: _.sage()
a + 1
```

LibGAP Class Hierarchy

```
$ grep 'class GapElement' sage/libs/gap/element.pxd
cdef class GapElement(RingElement):
cdef class GapElement_Integer(GapElement):
cdef class GapElement_Rational(GapElement):
cdef class GapElement_IntegerMod(GapElement):
cdef class GapElement_FiniteField(GapElement):
cdef class GapElement_Cyclotomic(GapElement):
cdef class GapElement_Ring(GapElement):
cdef class GapElement_String(GapElement):
cdef class GapElement_Boolean(GapElement):
cdef class GapElement_Function(GapElement):
cdef class GapElement_MethodProxy(GapElement_Function):
cdef class GapElement_Record(GapElement):
cdef class GapElement_RecordIterator(object):
cdef class GapElement_List(GapElement):
cdef class GapElement_Permutation(GapElement):
```

Function Factory

```
sage: cyc = \
....:   libgap.function_factory('CyclicGroup')
sage: cyc(4)
<pc group of size 4 with 2 generators>
sage: type(_)
<type 'sage.libs.gap.element.GapElement'>
sage: cyc(4).GeneratorsOfGroup()
[ f1, f2 ]
sage: g1, g2 = _
sage: g1^2
f2
sage: g1^3
f1*f2
```

TODO

- Signal handling
- Would like better instrumentation for GASMAN, e.g. who keeps object alive?
- Being able to reallocate the GAP memory pool would be nice
- No global variables would be nice, too
- Provide sane C++ API on top of libGAP?
- Upstream it?

Thanks

Both GAP and Singular have been extremely useful for my own research in string theory.

Thank you for writing it and making the code available to us all!

title