Using libpolymake.so

polymake–workshop
Darmstadt 2011

Benjamin Lorenz

Goethe–Universität Frankfurt
blorenz@math.uni-frankfurt.de

01.04.2011
polymake structure

The usual polymake consists of

- a (small) perl script polymake
- which loads several perl modules for
  - managing (polymake-)objects, properties, rules
  - the shell, the scheduler
  - and some more technical stuff
- lots of .rules files (parsed by the perl modules)
- several shared libraries (.so) for
  - hacking into the perl interpreter
  - the C++-clients of each application (including common)

libpolymake consists of

- one shared library libpolymake.so which is linked against
- libperl.so to load all perl modules and then the rule-base
- which again load the other shared libraries of all applications
Using libpolymake in five short steps

- `#include <polymake/Main.h>`
- initialize polymake by creating an instance of `polymake::Main`
- set an application
- work with polymake like in any C++ client (see PTL,CPP)
- link it against `libpolymake.so` and few other libraries
```cpp
#include <polymake/Main.h>
#include <polymake/Matrix.h>
#include <polymake/SparseMatrix.h>
#include <polymake/Rational.h>
using namespace polymake;

int main(int argc, const char* argv[]) {
    try {
        const int dim = 4;
        Main pm;
        pm.set_application("polytope");
        perl::Object p("Polytope<Rational>");
        p.take("VERTICES") << (ones_vector<Rational>() |
            3*unit_matrix<Rational>(dim));
        const Matrix<Rational> f = p.give("FACETS");
        const Vector<Integer> h = p.give("H_STAR VECTOR");
        cout << "facets" << endl << f << endl << "h*" << h << endl;
    }
    catch (const std::exception& ex) {
        std::cerr << "ERROR: " << ex.what() << endl; return 1;
    }
    return 0;
}
```
polymake::Main

Main(user-settings = "user")

*The constructor for Main has one optional argument which specifies if polymake should load user-settings (usually from ~/\.polymake). Other possible values are "none" or a path to a configuration directory.*

main.set_application("appname")

*Sets the current application and loads the corresponding data if neccessary.*

More methods:

- set_application_of(Object)
- add_extension("dir"), include("rule_file")
- set_preference("label"), reset_preference("label")
- get_custom("name"), set_custom("name", value), reset_custom("name")
polymake::perl::Scope

- corresponds to one input line in the shell
- used for some cleanup, e.g. removing temporary properties created from Main via main.newScope()
- need to be properly nested
- provides methods to temporary set preferences and custom variables:
  - `prefer_now("label")`
  - `set_custom("name", value)`
Building your program

There is a small tool `polymake-config` installed side by side with the main `polymake` script, which tells you everything necessary:

usage: polymake-config --help | --version | [--debug] --OPTION

Print bits of polymake configuration useful to compose Makefiles for programs linked with its callable library.

OPTION may be one of:

- `--cc` print the name of C++ compiler and linker
- `--cflags` print the C++ compiler flags without header paths
- `--includes` print the C++ compiler flags for header paths
- `--ldflags` print the linker flags
- `--libs` print the libraries to link with

Some notes:

- in `cflags` only `-DPOLYMAKE_DEBUG={0,1}` and `-fPIC` is obligatory
- in the workshop version 2.9.10 you need to add `-lxml2` to the linker