Flexible Object Hierarchies in `polymake`

Ewgenij Gawrilow, Michael Joswig

Depts. of Mathematics at
Technische Universität Berlin &
Technische Universität Darmstadt

ICMS 2006 – Castro Urdiales
September 2, 2006

Project History

- Initiated 1996 by Prof. Günter M. Ziegler, chair for Discrete Geometry, Berlin, as a collection of tools for studying polytopes (convex polyhedra).
- Since 1997 in co-authorship with Michael Joswig
- with many contributions from Thilo Röhrig, Nikolaus Witte, Marc Pfetsch, Volker Kaibel, Axel Werner, et al.
- First presented at ICM 1998, Berlin
- Fully free, open source software under GPL

`www.polymake.de`

Available in source code form and as Linux RPM; included in FreeBSD port collection and MacOS X Fink project.

Project Goals

- Unified user interface to a variety of free software tools available in the field (a plenty of mutually incompatible file formats, command-line options, etc.)
- Providing means for visualization from different aspects (geometric, combinatorial, ...)
- Flexibility on many levels: user preferences, ease of interfacing of new tools, self-made extensions: C++ API, Template Library
- *rapid prototyping* - perl as scripting language for repetitive tasks like searching for special properties in a large family of objects

Modularity

The core system is almost "math-free", the whole mathematical know-how is bundled in *applications* ($\cong$ packages).
An application is a collection of object types, atomic property types, algorithms, graphical primitives, visualization methods, and other top-level user functions, as well as programming aids like C++ libraries or perl modules.

- `polytope` – convex polytopes and unbounded polyhedra in arbitrary dimensions
- `topaz` – finite simplicial complexes
- `surface` – polyhedral surfaces (currently in 2-D only)
- `tropical` – tropical polytopes

Applications may import everythings from each other (inheritance) or reuse some definitions (aggregation).

polytope

Main object type: `Polyhedron`: over 100 geometric and combinatorial properties.

- convex hull computation ($\mathcal{V} \leftrightarrow \mathcal{H}$ representation), vertex graph (skeleton), face lattice, triangulations ...
- linear and abstract objective functions, integral lattice points, Steiner points ...
- visualization in 2-D, 3-D, and 4-D (Schlegel diagrams), graphs and face lattice, Gale diagrams ...
- transformations, projections, truncation, lifting, stacking ...

Special cases: zonotopes, Voronoi diagrams, tight spans, rigid frameworks

Classical Objects vs. polymake

How would an implementation in a classical OO-language (C++, Java) look like?

```
class Polyhedron {
   Matrix vertices;     boolean simplicial;
   Matrix facets;       FaceLattice face_lattice;

   // constructors
   Polyhedron(Matrix vertices);
   Polyhedron(Matrix facets);

   // data access
   boolean getSimplicial() const;
   FaceLattice getFaceLattice() const;
}
```

Problems

- Each constructor should compute every known property, otherwise the object state would be instable
  $\Rightarrow$ Solution: access functions deploy lazy evaluation
- Introducing independent new features:
  Someone wants to add `FLAG_VECTOR`, the second wants to add `N_LATTICE_POINTS`

```
class Polyhedron_with_FlagVector
  : public Polyhedron { ... }
class Polyhedron_with_LatticePoints
  : public Polyhedron { ... }
```

  The third wants to use the both extensions. But multiple inheritance is not viable!

Open Object

The definition scope of an object type in `polymake` can be reopened as often as needed, even spread over multiple source files. Core description:

```
object Polyhedron {
  property VERTICES : Matrix;
  property FACETS : Matrix;
  property SIMPLICIAL : Boolean;
}
```

First extension:

```
object Polyhedron {
  property FLAG_VECTOR : Vector;
}
```

Lazy Evaluation

`polymake` objects usually start with minimal information needed to uniquely identify the chosen equivalence class of mathematical objects. The most properties are computed on demand, using *production rules*.
A production rule is an algorithm adorned with declarations of the source properties (inputs) and target properties (results):

```
FACETS, VERTICES_IN_FACETS : VERTICES | POINTS

SIMPLICIAL : VERTICES_IN_FACETS
```

Design guideline: try to keep each computational step as short as possible, but without losing efficiency and/or useful by-products.

Rule Scheduling

Being asked for an object property, `polymake` tries to find an applicable chain of production rules, including, probably, several intermediate steps.
Along with the main information about source and targets, it uses some optional meta-data describing the rules:

- `WEIGHT` nested rule delivering a coarse measure of the computational complexity (linear-time, polynomial, super-polynomial)
- `PRECONDITION` nested rule tells whether the rule is applicable to the given object.
- groups of similar production rules can be ordered by user's preferences, so that the favourite one is always tried first

The rule scheduling then boils down to a Dijkstra-like shortest path search w.r.t. rule weights.

Object Consistency

Why is it safe to let everybody reopen and extend the object definitions?

Production rules are only object methods that are allowed to add new computed properties to the object. Neither rules nor other methods are allowed to overwrite existing object properties. `polymake` objects are, in fact, immutable.

This approach resembles the usual mathematical way of expression. "Let P be a polytope ..." means P stays the same until its scope (theorem, paragraph, etc.) ends.

Object Persistence

The object properties are considered "valueable" and expensive to compute. Therefore `polymake` keeps all properties, including those needed in intermediate steps, after the computation is finished. If the object is going to be stored in a data file, all its properties go with it.
This allows to avoid re-computations if the object properties are being queried later again.

Some properties, however, can be declared *temporary*. `polymake` gets rid of them as soon as possible. Usually these are either very trivial properties, or ones requiring huge amount of space to be stored.

Property Types

Property types are declared in the object or application scope, in the last case shareable between different object types.

- *atomic* properties like Scalar, Vector, Matrix, IncidenceMatrix. They are opaque from the `polymake` core's point of view. Many complex types are implemented in C++ and have an efficient perl interface.
- *subobject* properties take other `polymake` objects as values. For example, the triangulation of a polytope is a special case of a finite simplicial complex, defined in the application `topaz`.

```
property VERTICES : Matrix;
property TRIANGULATION : topaz::SimplicialComplex;
```

Unique and Multiple Properties

The cardinality of occurance in the object.

- *unique* property is allowed to occur at most once. It's the default behaviour, suitable for the most of the properties: `VERTICES`, `FACETS`, etc.
  This does not imply the uniqueness of representation: the set of `VERTICES` can be arbitrarily permuted, but it still describes the same polytope.
- *multiple* properties can be declared in cases when some property can take substantially different values, each of them being potentially interesting for further research. A polytope can be triangulated in very many ways, some of them having special features. Thus it is allowed to possess several TRIANGULATION instances.

Rules for Subobjects

Production rules may have the subobject's properties as both their sources and targets. The set of available production rules for an object is automatically augmented by the suitable rules defined in the containing object. This can be seen as a kind of dynamic class derivation.

```
object Polyhedron {
   TRIANGULATION.FACES : VERTICES
   VOLUME : TRIANGULATION.FACETS, VERTICES
}
```

Each subobject always knows its containing object via the special `parent` property.

Mutable Subobjects

Can be attached to an object without changing its semantics, mainly to gain profit from the rule scheduling. Example: a linear programming problem applied to a polytope – a mutable mulitple property.

```
object LinearProgram {
  property OBJECTIVE : Vector;
  property MAXIMAL_VALUE : Scalar;
  property MAXIMAL_FACE : Set;
}
object Polyhedron {
  property LP : LinearProgram;
  LP.MAXIMAL_VALUE, LP.MAXIMAL_FACE : \
    LP.OBJECTIVE, VERTICES, GRAPH
}
```

Object Derivation Revisited

`polymake` does not use object derivation for expanding its capabilities. Instead, deriving from an object type always expresses some narrowing of the modelled domain.

- narrower equivalence classes: an instance of `CombinatorialPolytope` represents all polytopes combinatorially isomorphic to each other. The vertex-facet incidence information suffices to get all combinatorial properties.

```
object CombinatorialPolytope {
  property VERTICES_IN_FACETS;
}
object Polyhedron : CombinatorialPolytope {
  VERTICES_IN_FACETS : VERTICES, FACETS
}
```

Object Derivation, continued

- Special cases: Some properties are only defined for restricted families of polytopes. Some special constructions may be comfortably encoded this way, too.

```
object Zonotope : Polyhedron {
  property INPUT_VECTORS : Matrix;
  VERTICES, FACETS : INPUT_VECTORS
}
object VoronoiDiagram : Polyhedron {
  property SITES : Matrix;
  FACETS : SITES
}
```

C++ binding

How to access these funny objects from "real" programs?

```
polymake::Object polarize(polymake::Object& p) {
   polymake::Object q("Polyhedron");
   Matrix<Rational> V;
   p.give("VERTICES") >> V;
   q.take("FACETS") << V;
   return q;
}
```

The programmer has (almost) free choice of data structures where to store the properties. The only requirement: the C++ class should be structurally equivalent to the declared type of object property. (Sometimes involves template metaprogramming.)

Conclusions

`polymake` object model:

- promotes the modularization of software
- comes close to the mathematical notion of objects
- allows for flexible extensions of property sets and inheritance hierarchies, even dynamically at runtime
- allows the programmer freedom in choice of data structures
- is aimed primarily for exact algorithms on discrete structures, not suitable for numeric applications, approximation, or stochastic algorithms
- involves an overhead for an interpreted language, not a pure C++ solution