C++ Concepts for Ill-posed Inverse Problems

Or: How to do weird math stuff at compile time

David Frank  --  frankd@in.tum.de

Complete code at https://godbolt.org/z/EBuSA5
What we'll be doing

Convert key ideas of elsas* design, which currently does a lot at runtime, to work at compile time

* You can find it at: https://gitlab.lrz.de/IP/elsa
What is an ill-posed Inverse Problem

- Arise when, we want to compute information about hidden data from outside
- Can represent image processing or tomographic reconstruction task
- Check out the book *Discrete Inverse Problems* by Per Christian Hansen (Math heavy!!)
What is an ill-posed Inverse Problem

- Arise when, we want to compute information about hidden data from outside
- Can represent image processing or tomographic reconstruction task
- Check out the book *Discrete Inverse Problems* by Per Christian Hansen (Math heavy!!)

The concept Problem

- Models mathematical optimization problem
- Incorporates logic, so an algorithm can find a solution (called a Solver)

Currently *elsa* does all checks, if the *Solver* can actually handle the problem at run time.
A base concept for a problem

```
template<class P_>
class Problem = requires(P_ p) {
    { p.dataTerm_ } -> Functional;
    requires requires (vector x) {
        { p.gradient(x) } -> convertible_to<vector>;
    };
};
```

How to read this:

If a type has a member `dataTerm_`, which fulfills the concept `Functional` and has a member function `gradient`, which takes a `vector` and returns something, which is convertible to a `vector`, it satisfies the concept `Problem`
Specialized Problems

For example, *Conjugate Gradient* requires a data term in quadric form

- Introduce concepts, which require the original Problem and impose further restrictions

Specialized Problems

For example, *Conjugate Gradient* requires a data term in quadric form

- Introduce concepts, which require the original Problem and impose further restrictions

Concepts for *Conjugate Gradient*

```cpp
template<class P_>
concept QuadricProblem = Problem<P_> && requires(P_ p) {
    requires is_specialization< decltype(p.dataTerm_), quadric>;
};

template<class P_>
concept ConvertibleToQuadricProblem = Problem<P_> && requires(P_ p) {
    requires
        is_specialization< decltype(p.dataTerm_), l2_norm_pow2> &&
        requires { { p.toQuadric() } -> QuadricProblem; }
};
```

* for the interested https://en.wikipedia.org/wiki/Conjugate_gradient_method
A promising approach for elsaf*:

- Remove flat polymorphic hierarchies
- Remove coupling between some components
- Describing mathematical constrains at compile time

Personally, the approach of *requiring something* feels natural

* and maybe for your project?
I hope you enjoyed my talk!

David Frank  --  frankd@in.tum.de

Complete code at https://godbolt.org/z/EBuSA5