Design: Dynamic AST Matchers

This document contains a design proposal for a dynamic AST matcher layer that supports creating arbitrary matchers at runtime, which will allow for generic tools to be developed.

Context

The existing AST matcher framework provides a very powerful and compact way to traverse the AST and find specific nodes by providing complex predicates.
However, this framework requires that all predicates to be used at runtime are fully defined at compile time. The creation of these predicates can be a big challenge in some cases, and this limitation forces a ‘recompile’ step on this creative loop.

Also, without being able to specify matchers at runtime we can’t support more generic search use cases. For example, an editor-based search tool that supports an AST matcher predicates.

Goals

- Runtime parsing of matcher expressions, including literals and predicates. Feedback for syntax/semantic errors in the expressions

- Create matchers at runtime by name, with dynamically created arguments if required by the matcher
- Query command line tool that runs the matcher on a specific compilation unit and reports all matches

Non-goals

- Dynamic refactoring tools. They might be implemented using dynamic matchers, but it is out of the scope for this library
- Full C++ expression support for predicates. Values are going to be limited to literals (eg. 1, “foo”). Expressions like 1+2 will not be parsed. Some useful constants (like INT_MAX) might be available

Code location

The main output of the project will be a library to be used by AST matcher related clang-tools, and should live close to the already existing AST matcher library, but as a separate library

Matcher implementation

One aspect that simplifies the implementation of dynamic matchers is to have a uniform generic signature for creating them. Otherwise, whatever code that creates them will require explicit knowledge of each matcher, or at least of each signature.
The generic signature is of the form `VariantType (*)(const std::vector<VariantType>& args)`.
A variant type instance can contain numbers, booleans, strings and Matchers, as well as error related information for return values.
Instead of reimplementing all matchers to have a generic signature, we implement marshaller functions that provide the generic signature on top of the regular static one. The marshallers verify that the argument count and types are correct and calls the underlying constructor function.

Registry

At initialization of the framework, all known matchers are added into a registry. This registry provides a way to create matchers at runtime by name.
This registry is the only component that would require maintenance when the ASTMatcher library is updated. Any new matcher, and new overloads for existing matchers, would have to be added to the registry. In most cases, a single line of code per matcher does the job.

Matcher expression parsing

A simple matcher expression parser, combined with the matcher registry, allows clients to go from a user provided string to a `Matcher<T>`. The output of the parser is a variant type also, which allows it to return Matchers as well as syntax and semantic errors.
The parser interface provides a way for the client to inject and/or modify the matcher tree as it is being created. For example, a client could insert `Id()` matcher around specific nodes without having to do it in the string form.