Compile-time 'reparsing'

Putting a Compiler Inside Your Compiler

Gordon Woodhull
C++Now 2012

https://svn.boost.org/svn/boost/sandbox/metagraph/boost/metagraph/angly

Goal: Static Graph EDSL

```
typedef graph<node<A, edge<t, B>, edge<u, C> >,
              node<B, edge<v, D> >,
              node<C, edge<w, E> >,
              node<D, edge<x, F> >,
              node<E, edge<y, F>, edge<z, G> > >
adjacencies;

typedef typename angly::run_dfa<
  typename mpl::first<graph_parser>::type,
  graph_nodes,
  adjacencies,
  mpl::at<typename mpl::second<graph_parser>::type,
          mpl::_1> >::type
  graph_data;

typedef typename
  mpl_graph::adjacency_list_graph<graph_data> parsed_graph;
```

So you want an EDSL

- Get a metaprogram to reparse the code
- Expression ASTs – Proto
- Strings – metaparse
- Bracket expressions - angly (roundy)
- Preprocessor metaprogramming

Graph-Parsing Automaton

```
typedef angly::dfa<
  angly::state<graph_nodes, mpl::true_, mpl::vector<>,
    angly::transition<graph_node, graph_nodes,
                 angly::match_token<node<>>, node_name,
                 mpl::push_back<mpl::_1, mpl::_2>>>,
  angly::state<node_name,
    angly::transition<node_name_trans, node_edges,
                 mpl::always<mpl::true_>, void,
                 mpl::pair<mpl::_2, mpl::vector<>>>>,
  angly::state<node_edges, mpl::true_,
    angly::transition<node_edge, node_edges,
       angly::match_token<edge<>>, edge_name,
       mpl::pair<mpl::first<mpl::_1>,
          mpl::push_back<mpl::second<mpl::_1>, mpl::_2>>>>,
  angly::state<edge_name,
    angly::transition<edge_name_trans, edge_target,
                 mpl::always<mpl::true_>, void, mpl::_2>>,
  angly::state<edge_target,
    angly::transition<edge_target_trans, edge_done,
                 mpl::always<mpl::true_>, void,
                 mpl::pair<mpl::_1, mpl::_2>>>,
  angly::state<edge_done, mpl::true_>
> graph_dfa_desc;
```

Automaton-Parsing Automaton (eek!)

```
struct dfa_a_state :
    dfa_transition<match_token<state<>>,
       state_name,
       mpl::pair<mpl::push_back<mpl::first<mpl::_1>,
                   mpl::pair<s_name<mpl::_2>,
                             s_transitions<mpl::_2>>>,
          mpl::push_back<mpl::copy<s_stmap<mpl::_2>,
                mpl::back_inserter<mpl::second<mpl::_1>>>,
                         mpl::pair<s_name<mpl::_2>,
                      detail::apply_seq_q<dfa_state>
                      ::apply<s_args<mpl::_2>>>>>>> {};
typedef boost::msm::mpl_graph::adjacency_list_graph<
mpl::vector<mpl::pair<dfa_states,
  mpl::vector<mpl::pair<dfa_a_state, dfa_states>>>,
mpl::pair<state_name,
  mpl::vector<mpl::pair<state_name_trans, state_finish>>>,
mpl::pair<state_finish,
  mpl::vector<mpl::pair<state_name_transition, state_transition
              mpl::pair<state_finish_trans, state_data>>>,
mpl::pair<state_data,
  mpl::vector<mpl::pair<state_finish_transition, state_transiti
              mpl::pair<state_data_trans, state_transitions>>>,
                ...
>> dfa_dfa;
```

Chomsky's Grammar Hierarchy

Type	Grammar	Machine
0	recursively enumerable	Turing machine
1	context-sensitive	linear-bounded TM
2	**context-free**	**pushdown automaton**
3	regular expressions	finite state automaton

Proto, metaparse, and angly all parse **Type 2** grammars

Recursive descent vs. pushdown automaton

- Equivalent computation
- Use the compiler's stack vs. explicit stack
- Backtracking allows non-determinism
- Proto, metaparse – backtracking, implicit stack
- angly (roundy) – deterministic, explicit stack

Proto

```
struct MapListOf
  : proto::or_<
        proto::when<
            proto::function<
                proto::terminal<map_list_of_tag>
              , proto::terminal<_>
              , proto::terminal<_>
            >
          , insert(
                proto::_data
              , proto::_value(proto::_child1)
              , proto::_value(proto::_child2)
            )
        >
      , proto::when<
            proto::function<
                MapListOf
              , proto::terminal<_>
              , proto::terminal<_>
            >
          , insert(
                MapListOf(proto::_child0)
              , proto::_value(proto::_child1)
              , proto::_value(proto::_child2)
            )
        >
    >
{};
```

```
std::map<std::string, int> op = map_list_of("<", 1)("<=",2)(">", 3);
```

Metaparse

```
/* expression ::= plus_exp
 * plus_exp ::= prod_exp ((plus_token | minus_token) prod_exp)*
 * prod_exp ::= int_token ((mult_token | div_token) int_token)*
 */
typedef
  foldlp<
    sequence<one_of<mult_token, div_token>, int_token>,
    int_token,
    eval_mult
  >
  prod_exp;

typedef
  foldlp<
    sequence<one_of<plus_token, minus_token>, prod_exp>,
    prod_exp,
    eval_plus
  >
  plus_exp;

typedef last_of<any<space>, plus_exp> expression;
```

```
apply_wrap1<calculator_parser, _S(" 1+ 2*4-6/2")>::type::value
```

Taking Apart Templates

```
template<typename T> struct de_arg { typedef void type; }
template<template<typename...> class Template,
         typename ...Args>
struct de_arg<Template<Args...> > {
  typedef Template<> type;
};

template<typename ...Args> struct variadic_to_mpl;
template<> struct variadic_to_mpl<> : mpl::vector<> {};
template<typename Head, typename ...Args>
struct variadic_to_mpl<Head, Args...>
  : mpl::push_front<typename variadic_to_mpl<Args...>::type,
                    Head> {};

template<typename T> struct arg_seq;
template<template<typename...> class Template,
         typename ...Args>
struct arg_seq<Template<Args...> >
  : variadic_to_mpl<Args...> {};
```

Onward and Inward...

```
template<typename DFA, typename StartState, typename Input,
         typename PropFn>
struct dfa_sequence {
  typedef typename detail::de_arg<Input>::type token;
  typedef typename detail::arg_seq<Input>::type inner_seq;
  typedef typename detail::create_stack<
    typename get_state_start_data<typename PropFn::template
                  apply<StartState>::type>::type,
    StartState,
    typename mpl::begin<inner_seq>::type,
    typename mpl::end<inner_seq>::type>::type stack;
  typedef detail::dfa_iter<DFA, stack, PropFn> begin;
  typedef detail::dfa_end end;
};
template<typename DFA, typename StartState, typename Input,
         typename PropFn = mpl::_1>
struct run_dfa :
  mpl::fold<dfa_sequence<DFA, StartState, Input,
                    typename mpl::lambda<PropFn>::type>,
            void,
            mpl::_2 > {}; // iterate until done
```

Iterators

```
template<typename Stack>
struct stack_is_done :
  boost::is_same<
    typename get_frame_state<
      typename mpl::front<Stack>::type>::type,
    utterly_done_state>
{};

struct dfa_end {};

template<typename DFA, typename Stack, typename PropFn>
struct dfa_iter {
  static_assert(!stack_is_done<Stack>::value,
                "iterate past end of DFA sequence");
  typedef typename dfa_step<DFA, Stack, PropFn>::type eval;

  typedef typename
  mpl::if_<typename stack_is_done<eval>::type,
           dfa_end,
           dfa_iter<DFA, eval, PropFn> >::type next;
  typedef typename get_frame_data<
    typename mpl::front<eval>::type>::type type;
};
```

Creation

```
struct utterly_done_state {};

template<typename Data, typename StartState,
         typename CurrIter, typename EndIter>
struct create_stack {
   typedef mpl::vector<typename
                      create_frame<Data,
                                   StartState,
                                   CurrIter,
                                   EndIter>::type,
                      typename //end-frame
                      create_frame<void,utterly_done_state,
                                   void, void, mpl::_2>::type
                      > type;
};
```

Interlude: Graph Metaprogramming

- “Graphs are more common in metaprogramming than in runtime programming”
  - Control flow, data flow, call graphs, parallelization
  - Pointers, component configuration, schemas
  - State machines
  - Class hierarchies
  - Ownership, containment
  - Grammars, abstract syntax trees, expression trees

Analyzing & Generating Code from Graphs

- A class node encapsulates any program element: object, code, state, rule
- Edges represent the relations
- Graph metadata can be used to
  - prove things about code
  - structure code better
  - generate complex programs robustly

Graph Metadata Languages

- Nodes and edges are types
- Text only allows hierarchical tree structure
- Cross references make it graphy
- Most are more than vanilla node/edge graphs
    - typed edges / nodes
    - multiple identities (node is an edge is a graph)
    - easy for text to represent

A Single Step

```
template<typename DFA, typename Stack, typename PropFn>
struct dfa_step {
  typedef typename mpl::front<Stack>::type frame;
  typedef typename get_frame_curr_iter<frame>::type
      curr_iter;
  typedef typename get_frame_end_iter<frame>::type
      end_iter;

  typedef typename
  mpl::eval_if<typename boost::is_same<curr_iter,
                                       end_iter>::type,
             finish_frame<Stack, PropFn>,
             dfa_follow<DFA, Stack, PropFn>
             >::type type;
};
```

Transition

```
template<typename DFA, typename Stack, typename PropFn>
struct dfa_follow {
  typedef typename mpl::front<Stack>::type frame;

  typedef typename get_frame_curr_iter<frame>::type curr_iter;
  typedef typename mpl::deref<curr_iter>::type item;
  typedef typename match_item<DFA,
        typename get_frame_state<frame>::type, item, PropFn
  >::type match;
  typedef typename
  get_transition_recurse_rule<typename
      PropFn::template apply<match>::type>::type recurse_rule;
  typedef typename mpl::eval_if<
    typename boost::is_same<void, recurse_rule>::type,
    dfa_follow_no_recurse<DFA, Stack, match, item, PropFn>,
    dfa_follow_recurse<DFA, Stack, recurse_rule,
                       match, item, PropFn> >::type type;
};
```


18

No Recurse

```
struct dfa_follow_no_recurse {
  typedef typename mpl::front<Stack>::type frame;
  typedef typename mpl::pop_front<Stack>::type popt;
  typedef typename
  mpl::push_front<popt, typename create_frame<
   typename mpl::apply<typename get_transition_post_action<
     typename PropFn::template apply<Match>::type>::type,
              typename get_frame_data<frame>::type,
              Item>::type,
   typename msm::mpl_graph::target<Match, DFA>::type,
   typename mpl::next<
        typename get_frame_curr_iter<frame>::type>::type,
   typename get_frame_end_iter<frame>::type>::type
  >::type type;
};
```

Recursing 1

```
template<typename DFA, typename Stack, typename RecurseRule,
         typename Match, typename Item, typename PropFn>
struct dfa_follow_recurse {
  typedef typename mpl::front<Stack>::type frame;
  typedef typename mpl::pop_front<Stack>::type popt;
  typedef typename
  mpl::push_front<popt, typename create_frame<
   typename get_frame_data<frame>::type,
   typename msm::mpl_graph::target<Match, DFA>::type,
   typename mpl::next<
       typename get_frame_curr_iter<frame>::type>::type,
   typename get_frame_end_iter<frame>::type,
   typename get_transition_post_action<
     typename PropFn::template apply<Match>::type>::type>::type
  >::type parent;
```

Recursing 2

```
  typedef typename detail::arg_seq<Item>::type inner_seq;
  typedef typename
  mpl::push_front<parent, typename create_frame<
   typename mpl::apply<
     typename get_transition_pre_action<
       typename PropFn::template apply<Match>::type>::type,
     typename get_frame_data<frame>::type,
     typename get_state_start_data<
       typename PropFn::template apply<RecurseRule>::type
     >::type
   >::type,
   RecurseRule,
   typename mpl::begin<inner_seq>::type,
   typename mpl::end<inner_seq>::type>::type>::type type;
};
```


21

Popping Out

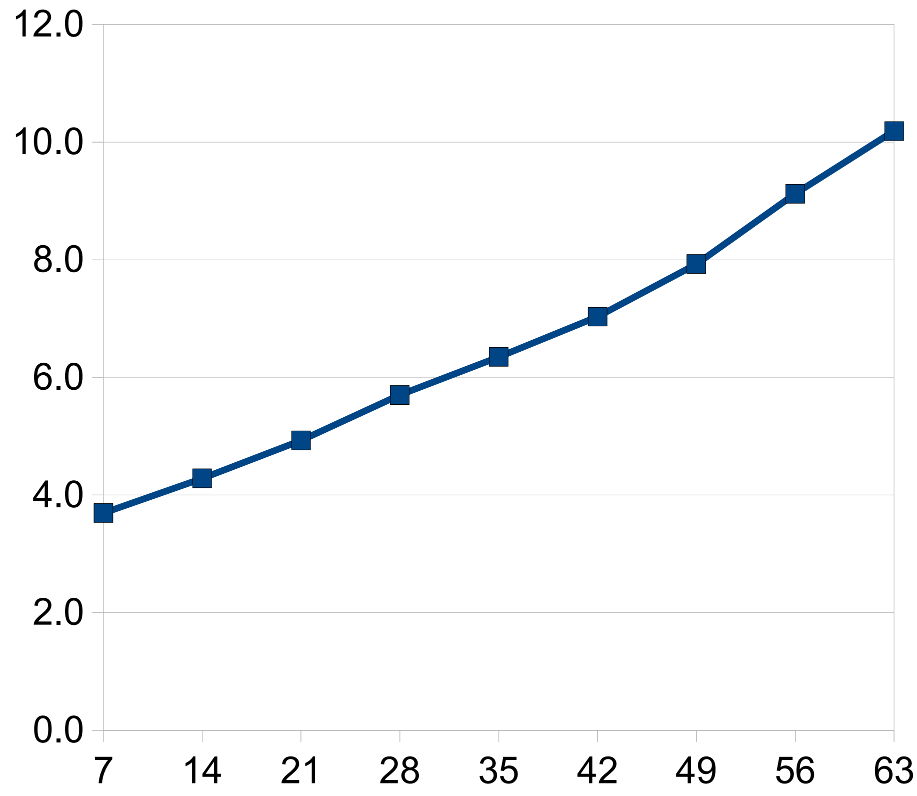
```
template<typename Stack, typename PropFn>
struct finish_frame {
  typedef typename mpl::front<Stack>::type frame;
  static_assert(is_finish_state<
     typename PropFn::template apply<
       typename get_frame_state<frame>::type>::type>::value,
                "must be in finishing state when input ends");

  typedef typename get_frame_data<frame>::type result;
  typedef typename mpl::pop_front<Stack>::type popt;
  typedef typename mpl::front<popt>::type frame2;
  typedef typename mpl::pop_front<popt>::type popt2;
  typedef typename
  mpl::push_front<popt2,
                typename
                set_data<frame2,
                         typename
      mpl::apply<typename get_frame_action<frame2>::type,
                typename get_frame_data<frame2>::type,
                result>::type>::type>::type type;
};
```

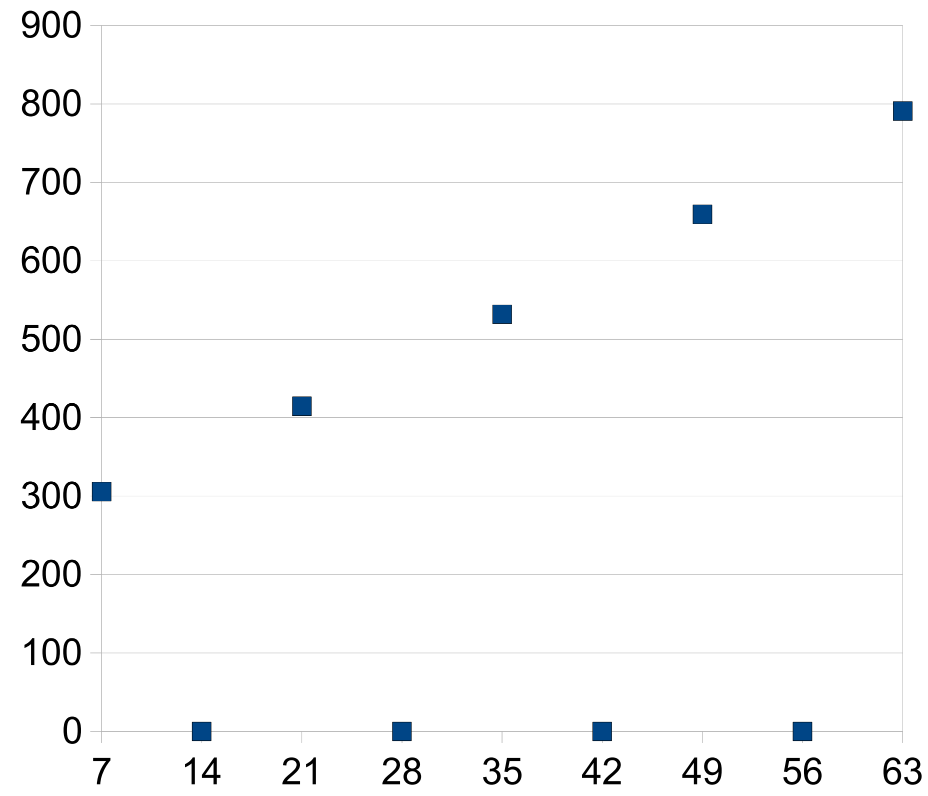

22

Performance

Parsing a graph with N nodes and N edges



Time, seconds



Memory, MB

gcc 4.6, 2 GHz Core i7, 16GB

23

Possible Futures

- Parse round brackets & mixtures
- Consider friendlier interfaces
- Generalize to general pushdown automaton
- Profile / improve performance
- Metagraphs!

24

Conclusion

Eventually we will replace the compiler with a metaprogram.

Conclusion

- If you ignore typename and ::template, TMP isn't really all that noisy
- (re)Parser designs converge
- Declarative languages on types are practical and fun