A Library for Graph Metaprogramming

Introducing MPL.Graph

Gordon Woodhull
dynagraph.org
gordon@woodhull.com

ABSTRACT

This paper describes the library MPL.Graph, which extends the Boost Metaprogramming Library into data structures and algorithms for graphs (networks).

Metaprogramming shifts calculations and abstraction penalties from runtime to compile time. Graph metaprogramming is applicable to a variety of problems where some or all of the graph structure is known at compile time, e.g. class hierarchies, state machines, expression trees, grammars, call graphs, ownership of objects, and pointers between objects.

A metaprogram can determine anything where the information is available at compile time, leaving the minimal code and data for runtime. In addition to this efficiency gain, this paper argues that metaprogramming provides clarity by cleanly separating what is known and what can be inferred at compile time versus what must be computed at runtime.

Categories and Subject Descriptors

D.2.13 [**Reusable Software**]: Domain engineering; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Data types and structures*

General Terms

Design

Keywords

Metaprogramming, Graph, C++

1. INTRODUCTION

Often it happens that we have some part of our data, such as parser rules or state machine transitions, at compile time.

Or there is a runtime algorithm, such as choosing which pointer to follow or which function to call based on a map or other indirection, that could be run at compile time.

Or there is structure in the code of a program that could be better expressed as data – the ways functions call each other, the ways objects get connected together, the way data flows.

All of these motivate metaprogramming, and in particular graph metaprogramming.

The Boost Metaprogramming Library (MPL) [1, 9] provides data structures and algorithms analagous to the Standard Template Library for defining and processing basic data structures at compile time. These *compile-time data structures* include vector, list, map, and set. Since a graph data structure is composed from exactly those data structures, it is fairly trivial to implement a compile-time graph data structure using MPL. MPL.Graph is less than 700 lines of C++ metafunctions.

This paper covers the implementation of MPL.Graph. But more interesting is the question, "Why would you want to do this?" This paper provides two concrete examples. In the current Boost release is a state machine checker for the Boost Meta State Machine Library [10], finding unreachable states and connected regions. The other is an experimental implementation of heterogenous runtime graphs with many vertex and edge types, called Fusion.Graph.

The Possibilities section describes many other places where graph metadata already exists, and how one might lift structure from program code into metadata by using compile-time graphs. This section is an invitation for involvement by the Boost and C++ community.

MPL.Graph came into existence to support a data structure called the *metagraph*, described in the last section of this paper. A metagraph is a system of objects produced by combining *patterns*, which are networks of concepts and their implementations. The metagraph will provide a data structure for all forms of higher-order graphs, and embodies a paradigm of *pattern-oriented programming*.

Finally, this paper argues that metaprogramming is more than just a trick for improving runtime efficiency: by expressing program structure as metadata, it provides a needed layer of abstraction that makes keeping track of complex code simpler and safer.

2. MPL.GRAPH DATA STRUCTURES

The MPL.Graph template classes and metafunctions implement concepts similar to those in the Boost Graph Library (BGL) [12], but translated from objects and ranges to types and type sequences.

Just as BGL implements runtime graph data structures in terms of the Standard Template Library's containers, MPL.Graph implements compile-time graph data structures by drawing on MPL's containers – in particular it makes ex-

BoostCon 2011 Aspen, CO USA

tensive use of `mpl::map` [1]. Vertices and edges are identified by unique types supplied by the user of the librqry.

Unlike with BGL, it is not necessary to choose which concepts the compile-time graph will implement upon instantiating the graph. Instead, when each concept is invoked, the library produces the appropriate metadata in $O(E)$ to handle all metafunctions in that concept in constant time. E.g. the first time `mpl::graph::in_edges<G, V>::type` is called, the library builds a two-layerd target to edge to source map from the input adjacency or incidence list, and it is not built if it is not used. Because of this design, there is little difference between MPL.Graph's `incidence_list` and `adjacency_list` in performance guarantees.

BGL has a functional interface in order to allow efficient and correct adaptation other libraries to its concepts. So the metafunctions in MPL.Graph are very similar. For example, `source(g, e)` in BGL becomes `source<G, E>::type` in MPL.Graph. The BGL interface is well proven to allow many implementations of the same interface, and we also hope that this familiar interface will make the library easy to learn.

Mutable graphs are not yet supported; this would require returning a new graph, since metadata is always immutable. Instead of mutability, future union and difference operations are likely to be the most efficient way to build compile-time graphs from smaller graphs.

3. MPL.GRAPH ALGORITHMS

So far MPL.Graph implements the two most basic graph algorithms: Depth First Search and Breadth First Search. Again, the interfaces are closely based on the interface of the Boost Graph Library: the client passes in a visitor which is called by the traversal algorithm at key points. But this is not a visitor *object*; instead, it is a struct containing multiple metafunctions. Each metafunction returns a new visitor state along with the state needed by the traversal itself (the color map, and the queue in case of BFS).

Like BGL, MPL.Graph supports a variety of visitor actions: for example `depth_first_search` supports `discover_vertex`, `finish_vertex`, `tree_edge`, `back_edge`, and `forward_or_cross_edge`. The algorithm is derived directly from the definition in the BGL documentation [13].

4. FINDING GRAPH FEATURES IN META STATE MACHINE

MPL.Graph is currently distributed as a sublibrary of Meta State Machine (MSM), where it is used to determine submachine regions and find inconsistencies in the transition table.

The states are vertices and the transitions are edges in this graph. Then MSM submachines ("regions") are connected components within the graph identified by their start states.

In order to spare the user the effort of labeling every state with its region, MSM uses MPL.Graph to traverse the graph and determine the regions. A depth-first search from each start state identifies the region, and makes sure that regions don't intersect. The search can also flag unreachable states. It is a connected components algorithm with user-specified starting points, and we hope to optimize it and include it among MPL.Graph's algorithms.

[1]It is possible that an `mpl::vector` implementation would be more efficient, if less convenient. The same interface should support such an implementation.

5. A SIMPLE HETEROGENOUS GRAPH DATA STRUCTURE

Any system of objects with pointers or references between them can be thought of as a heterogeneous graph data structure, where the objects are the vertices and the pointers they hold are the edges. If the system is complex, specified at compile time by users of the library, and there are not virtual interfaces, then it may be fruitful to build the object definitions from a compile-time graph.

The proposed **Fusion.Graph** library takes an MPL.Graph description of a set of objects and the pointers or containers of pointers between them, and generates objects which implement the description.

This library also defines BGL-like heterogeneous graph concepts which allow arbitrary systems of objects to be adapted as Fusion Graphs.

6. POSSIBILITIES

This section outlines possible applications for MPL.Graph, which the author hopes the Boost and C++ community will develop. In all of these cases, an ad-hoc graph data structure already exists implicitly or explicitly, and MPL.Graph will make it more powerful by exposing it to metaprogramming.

6.1 Parallelization

Joel Falcou's Quaff [8, 7] implements algorithmic skeletons for parallelizing code. Skeletons are series-parallel (properly nested) patterns for control flow, which allow automatic parallelization without worrying about lower-level synchronization constructs in the code. For example, a `for` loop can automatically be split into multiple work items to be executed in parallel, and a series of work items can be scheduled to execute in parallel with another unrelated series. This is of the same power as OpenMP and the `cobegin/coend` statements of ALGOL 68. [2]

We will apply graph algorithms to the skeletons to find opportunities for automatic optimization. In addition, generalizing skeletons into general compile time graphs will allow more complex control flow, such as DAGs or even cyclic parallelization patterns.

6.2 Spirit grammars and ASTs as Graph Metadata

The Boost.Spirit parsing library generates a compile-time graph of function objects calling each other by tying uses of rules to their definitions.

In mainstream Spirit, the compile-time graph is not available to metaprogramming because the rule objects use type erasure to make it easy to construct the objects without having to specify complicated types; type erasure is used as an emulation of C++0x `auto`. The technique also allows rule objects to remain uninitialized so they can be referred to before they are defined.

Fortunately, there is a construct in Spirit called the "subrule"[4], which allows compile-time resolution of rules. Each

[2]See this survey [3] for a comparison of classic concurrent constructs; a general graph corresponds to fork/join in their taxonomy.

subrule has a different type, and Spirit ties references to definitions using these types. [3] Viewing the grammar as a compile-time graph could enable validation and transformation of the grammar, e.g. to detect an instance of left recursion or an empty loop before it sends the parser into a stack overflow. It may even be possible to implement other parsing algorithms based on the same grammar.

In addition, abstract syntax trees are heterogeneous-typed trees, where the parent-child node type relationship is generated from a compile-time graph (like a schema or regular expression) which is simpler than the grammar but follows a similar pattern. [4] MPL.Graph promises to make maintaining the correspondence between the graphs easier.

6.3 Data Flow and Complex Types from Call Graphs

To generalize on Spirit's recursive descent parser objects calling each other, let us consider function objects arranged in a compile-time graph. Function objects are represented as vertices, and runtime decisions and data flow [11] are represented by edges.

If the functions are templated on their input parameter types, complex types can be built from simple operations calling the next functor along the chosen edge with a more complex type derived from the input types. Cycles are allowed as long as they don't introduce a cyclic type dependency.

Much as Spirit can build complex synthesized attributes automatically based on a grammar, these polymorphic functions arranged in a graph can build type systems of complexity exponential to the number of runtime options. The graph allows objects not to know the precise type they are operating on, nor their place in the graph of functions.

A linear version of this technique is used in Dynagraph [14] to configure graph layout engines from string options, where applying an option may mean wrapping a layout engine in a translation or transformation engine. Each configurator can build on the last without knowing the actual type, because every one acts on graphs and layout engines of polymorphic type. Currently configurators need to explicitly ignore configurations that are incompatible because they are all in one linear sequence. If arranged in a graph, the configurators would see a configuration only if it is relevant to them.

6.4 EDSL Expression Trees and Graphs

Many Embedded Domain Specific Languages have graph-like features, from referring to common identifiers in different places in the syntax trees. Spirit's grammar rules are the canonical example: when a rule is referenced within another rule, it causes the trees to be joined, possibly causing cyles. Such graphs also arise in data flow, parallelization, and other languages with relations between objects.

We propose a common library over Proto for covering aspects of syntax having to do with graphiness. As Phoenix provides generalized lambda functionality for EDSLs, such a library could provide common graph functionality, merging expression trees which refer to each other into graphs.

[3]For examples of connecting rules by type instead of by object reference in a "simple Spirit," see Larry Evans' libraries using MPL [6] and Proto [5].

[4]These layered graphs with levels of detail are one of the motivations for the future Metagraph, briefly described below.

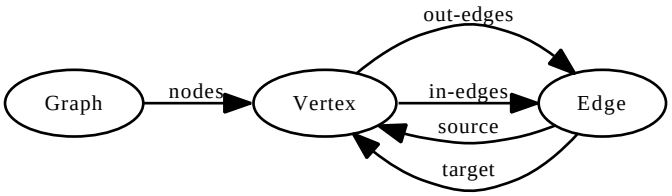

Figure 1: A traditional runtime graph data structure expressed as a graph of relations in a metagraph.

6.5 Shared pointer ownership

When objects contain shared pointers to each other, the object relation holds-reference-count-of forms a runtime graph, where the cycles can cause garbage collection problems. The common solution is to manually break cycles using `weak_ptr`, so that held reference counts form a DAG.

The compile-time part of this graph is classes with the relation can-hold-reference-count-to. If cycles exist in the class graph, a metaprogram could turn back-edges into weak pointers at compile time.

To take this further, imagine smart pointers which are aware of the full compile-time ownership graph. Many sophisticated ownership and garbage collection schemes could follow: the key is to view pointers as part of a pattern.

6.6 Schemas

Database and XML schemas also are graphs which are typically known at compile time. MPL.Graph could provide a structure for keeping track of the relations between rows or tags, and a way to traverse the data in a generic way.

7. THE METAGRAPH

The initial impetus for MPL.Graph is the author's quest for the "graph-of-graphs," the metagraph. Not (necessarily) related to the use of C++ template metaprogramming, the name "metagraph" refers to a data structure where a graph describes the topology of pointers between objects in a system.

The metagraph will represent subgraphs, hypergraphs, bipartite graphs, graphs at varying levels of detail, graphs within the vertices of graphs, clustered/nested subgraphs (where edges span levels), and generally, N-dimensional graphs as opposed to the conventional 2-dimensional graph of vertices and edges.

The metagraph starts with the heterogeneous graph described in section 5, and simplifies the construction and access of this data by building it out of overlaid *patterns*. A pattern is a graph of roles (objects) and the relations (container-like concepts) between them.

For example, a traditional runtime graph data structure can be described as in Figure 1: the graph contains a set of vertices, the vertex contains a set or two sets of edges, and an edge contains pointers to its two end vertices.

Each pattern is built on top of simpler patterns which it hides, combines, and refines. The simplest pattern – a pointer or a container of pointers – is represented by a single edge. A higher-level pattern can restrict and combine the semantics of multiple underlying patterns. For example, creating an edge means creating the edge object, connect-

ing its source and target pointers to the vertices, and adding the edge object to both vertices' edge sets; the graph pattern will also disallow direct setting of the edge's source and target. Similarly, when a vertex is deleted from a graph with subgraphs, the corresponding subvertices must be deleted from all subgraphs.

Maintaining such correspondences without formal patterns is messy and error-prone. The metagraph will make it possible to combine such patterns in complex combinations, with proper modularity.

8. RELATED WORK

As noted throughout this paper, there are numerous examples of ad-hoc compile-time graphs, such as Spirit's subrules, Proto's expression trees, MSM's transition tables, and Quaff's skeletons. Most of these have also implemented some form of depth-first search because it is the natural recursive way to traverse a graph.

Quite recently, the author learned of another compile-time graph library named Meta Graph Library, by Franz Alt [2]. It provides many of the same features as MPL.Graph, but provides a few more features, like undirected graphs and more algorithms layered on top of breadth-first search and depth-first search, like topological sort and cycle detection.

The authors will work together to combine the best features from the two libraries into a revised MPL.Graph. We have not yet benchmarked the performance of either, or fully compared the designs, but here are some preliminary observations:

MGL provides an iterator, rather than visitor, interface on its search algorithms. This reflects a classic design question in runtime graph libraries, which has no definitive answer. On the one hand, iterators are more convenient to the user of the library, because they don't require an inversion of control. On the other hand, visitors are more powerful because they can convey multiple events, whereas by design iterators convey only one. A combined library should support both interfaces.

MGL by design does not use recursion for its depth first search, instead managing a stack within the iterator metadata. This is probably safer on the compiler's stack than MPL.Graph's recursive DFS. (As usual, MPL.Graph's breadth first search is not recursive.)

Further analysis of how our designs converge or differ will result in an improved library.

9. CLARITY IN METAPROGRAMMING

C++ template metaprogramming is frequently maligned for making programs difficult to read and reason about. This is partly due to the accidental syntax, but it is also due to a lack of general understanding about what metaprogramming is for, how it works, and how to think about programs that run both at compile time and runtime.

This paper has shown a few examples of problems where shifting a responsibility from an implicit design or requirement in the code into a metaprogram, improves the correctness and clarity of the design. In particular there are many features of programming languages and code itself which are graphlike.

Metaprogramming can bring improvements in efficiency, but perhaps a greater benefit can be found in using it to generate the code or structure from a clearly described pattern. When program structure is mixed with runtime data structures and code, it is harder to see where the responsibilities lie. But if structure is described in metadata and a metaprogram generates the runtime data structures, design considerations become apparent and readily configurable.

10. ACKNOWLEDGMENTS

The author would like to thank Stephen North, Jason Eisner, Doug Gregor, Christophe Henry, and Joel Falcou for the encouragement to think about these interesting problems. The following Boost libraries form the basis and inspiration for the libraries described here: MPL by Alexey Gurtovoy and David Abrahams; Fusion by Joel de Guzman, Dan Marsden, Tobias Schwinger; BGL by Jeremy Siek, Doug Gregor, and many others.

11. REFERENCES

[1] D. Abrahams and A. Gurtovoy. *C++ Template Metaprogramming*. Addison-Wesley Publishing Company, New York, 2005.

[2] F. Alt. Meta Graph Library. `https://svn.boost.org/svn/boost/sandbox/mgl`.

[3] G. R. Andrews and F. B. Schneider. Concepts and notations for concurrent programming. *ACM Comput. Surv.*, 15:3–43, March 1983.

[4] J. de Guzman. Spirit Subrules. `http://www.boost.org/doc/libs/release/libs/spirit/classic/doc/subrules.html`.

[5] L. Evans. cfg_lookahead_extends Library. `http://www.boostpro.com/vault/index.php?&directory=Strings%20-%20Text%20Processing`.

[6] L. Evans. subrule_simple Library. `http://www.boostpro.com/vault/index.php?&directory=Strings%20-%20Text%20Processing`.

[7] J. Falcou and J. Sérot. Formal semantics applied to the implementation of a skeleton-based parallel programming library. *Parallel Computing*, 2008.

[8] J. Falcou, J. Sérot, T. Chateau, and J. Lapresté. Quaff: efficient C++ design for parallel skeletons. *Parallel Computing*, 32(7-8):604 – 615, 2006. Algorithmic Skeletons.

[9] A. Gurtovoy and D. Abrahams. The Boost MPL Library. `http://www.boost.org/doc/libs/release/libs/mpl/doc/index.htm`.

[10] C. Henry. The Boost Meta State Machine Library. `http://www.boost.org/doc/libs/release/libs/msm/doc/HTML/index.html`.

[11] S. Rajko. The Dataflow Library and the Arts. `https://github.com/boostcon/2008_presentations/raw/master/fri/DataflowlibraryArts.pdf`.

[12] J. Siek, L.-Q. Lee, A. Lumsdaine, et al. The Boost Graph Library. `http://www.boost.org/doc/libs/release/libs/graph/doc/index.html`.

[13] J. Siek, L.-Q. Lee, A. Lumsdaine, et al. The Boost Graph Library: Depth First Search. `http://www.boost.org/doc/libs/release/libs/graph/doc/depth_first_search.html`.

[14] G. Woodhull. Dynagraph. `http://www.dynagraph.org`.