GEN++ — an analyzer generator for C++ programs

Prem Devanbu
Articifial Intelligence Principles Research Department
prem@research.att.com

Laura Eaves
Object Oriented and Artificial Intelligence Technologies Group,
laurae@mozart.att.com

1 Introduction

The C++ programming language is becoming increasingly popular within and without AT&T. C++ supports an object-oriented style of programming, and has many powerful features that contribute to information hiding, software reuse, and program maintainability. Because of the increasing value of C++ software assets, it is important for developers and managers to have access to tools that extract information, generate metrics, check coding standards, etc. Tools such as CIA++ [3] make a significant contribution in this area, but more tools are needed. Unfortunately, tools for C++ are hard to build, primarily because of the complexity of processing the source—parsing, type checking, and symbol resolution for C++ are difficult tasks. In addition, introducing additional functionality into C++ tools, or tracking the evolution of the C++ language, involves major effort.

In this paper, we describe **gen++**, a tool generator for C++ 3.0. gen++ was implemented by attaching the GENOA/GENII [2] portable parse tree querying mechanism to the Cfront 3.0 compiler. gen++ is designed to reduce the difficulty of building C++ tools in 3 ways: first, tools are implemented by writing a specification in a compact domain specific language; second, all tools generated by gen++ use the Cfront compiler to do parsing, type checking and symbol resolution, and finally, gen++ masks Cfront's implementation details from the tool builder. To implement a particular C++ language tool with gen++, the tool builder simply specifies the operations on the C++ parse tree that are of interest to her; the specifics of processing the source language, and traversing the internal datastructures of Cfront, are no longer of concern.

2 Background–Decorated parse trees and GENOA/GENII

Language analysis tools (such as CIA, LINT, etc) can be thought of as having two phases. In the initial phase, the raw source is lexed, parsed, typechecked, all the symbols are resolved from uses to definitions, and an internal representation is constructed (this representation is called a "decorated parse tree", since it is a parse tree enhanced with notations for types, references, etc.). In the second phase, the decorated parse tree is traversed and the information relevant to the particular tool is extracted. In the case of CIA, the relevant information is definition/use occurrences of global names, macros etc; LINT, on the other hand, is extracts a range of information to check the proper use of types, values, expressions and so on. The important difference between different tools is *the processing they perform* *on the decorated parse tree*; thus the decorated parse tree forms the natural "domain of concern" for a language tool builder.

GENOA/GENII is a portable, language independent analyzer generator tool that can be used to create arbitrary analysis tools to process source files and extract useful information. An analysis tool is generated by writing a specification in the GENOA special-purpose query language. The query language has special traversal and iteration operators specially designed for processing decorated parse trees of programs, and is independent of any particular source language. GENOA is *portable*; it can use a decorated parse tree representation built by any language front end (that is implemented in C). It is ported to a new language front end by writing a specification in GENII. A GENII specification describes the data model of the decorated parse tree built by a particular front end, in a style similar to entity-relationship or semantic data modeling; it also describes the implementation of the decorated parse tree in the front end's specific data structures. This specification is compiled by GENII into a set of translation routines and look up tables. These translation routines and lookup tables support a standard, language- and front-end independent abstract datatype view (ADT) of decorated parse trees. This ADT is used by all the different tools generated by GENOA.

Thus, simply by writing a GENII specification, an existing front end can be turned into an analyzer generator. Other approaches to building language processing tools typically involve constructing a new front end from scratch. Very sophisticated compiler-generator tools have been proposed for constructing front ends, but given the syntactic and semantic vicissitudes of practical languages like C and C++, the GENII approach is still offers an economic advantage, if a front end is already available.

3 gen++ and its uses

`gen++` was constructed by simply interfacing GENOA to the Cfront compiler: a GENII interface was written, which describes the internal representation of a C++ decorated parse tree in the Cfront compiler. gen++ is fairly robust, and has been tested on it's own source code, and the source code for InterViews.

Using `gen++`, we have constructed a range of different tools, from analysis tools like CIA++ to metrics tools and coding standards checkers. To create tool with `gen++`, one writes an analysis specification in a domain-specific language. To illustrate, we show an example tool, which detects *when variables are modified, and when they are accessed*

```
ROOTPROC VarUse
PROC VarUse
ROOT Cfile;
{
 [
  (?NameRef
    (IF (OR (TYPEOF $parent UnaryCount) (AND (TYPEOF $parent Assignment) (EQUAL $slot lhs)))
        (THEN (PRINT stdout "Variable %s defined at %s" $token $location))
        (ELSE (PRINT stdout "Name %s accessed at %s" $token $location))))]
}
```

A specification in gen++ is a set of procedures, some of which are root procedures; these get invoked right after Cfront completes the construction of the decorated parse tree. In this case VarUse is the (only) root procedure. A procedure in gen++ is a series of *constructs*; a *construct* is an operation on a *current node*; it may print out the current node, copy it into a variable, or move to a child of the current node, etc. Nodes have *types*, which are elements of the C++ decorated parse tree—Expression, Declaration, Assignment etc are node types. Nodes have *slots*, the *fillers* of which are the children (or properties) of the current node. Thus, a node of type BinaryExpr has slots be_lhs and be_rhs, the filler of each is a node of type Expression. Some times, the filler of a slot is a list: for example, the (root) node, of type Cfile has a slot, globals, the filler of which is a list of Declarations in the file.

The various types of constructs are distinguished by enclosure in different types of parentheses. Let's look at the above procedure to illustrate the features of gen++. The ROOT Cfile declaration specifies the current node when the procedure gets invoked is a node representing the entire file. At this point, we conduct a global search of all the nodes below the root node ([...] is a global search), looking for nodes of type NameRef which are essentially references to names of any type. When we find name refernces, we check to see if the parent of that node is a unary increment or decrement operator (denoted by UnaryCount), or if the parent is an assignment and we got to the NameRef node via the be_lhs slot (which means the name reference being assigned to here). If one of these conditions is satisfied, it is a modification; otherwise it's just a use.

Many different tools have been implemented in gen++, to implement different tasks:

1. Generate an inheritance hierarchy of C++ classes.
2. Generate a graph of inter-classmember function calls.
3. Create a default output operator "<<" for a given class.
4. Generate control flow and data usage reports.
5. Generate C++ metrics: ratio of derived to base classes, ratio of number of public members to total number of classes, etc. (See Coplien [1] for a more complete list).
6. Check to see that destructors for base classes are always declared to be virtual.
7. Check to see that Constructors and Destructors are never declared virtual.
8. No member function should be both virtual and inline.

Note that the last 3 are coding standards checkers, and the others are information gathering tools. In most cases, the tools are on the order of 30-40 lines of gen++ specifications; the only exception in the list above is the control flow tool, which is about a 150 lines long, since it has to handle each kind of control construct in C++. Creating tools with gen++ is easy enough to consider building special-purpose, one-off tools to meet project-, or even subsystem-specific tool needs; we are certainly willing to do so on request.

4 Related Work

There are few related tools that work with C++. We have already discussed CIA++. The ALF/GRAIL system [4] includes a statically typed, rationalized, C++ source representation with a well defined application programmer interface. GENOA could be easily interfaced to ALF, to provide the same querying interface; the same advantages as **gen++** would accrue: a tool builder would only have to be familiar with the abstract structure of C++ programming language, as implemented in ALF; details of the data structures, classes, methods etc. of the ALF API would be hidden in the GENII interface specification. GENOA is simply a portable querying mechanism that can be attached to any representation of a parse tree.

5 Conclusion

gen++ is an analyzer generator for C++ programs. Several sample tools have been built with it. Documentation is available; the tool has been tested on different bodies of code, and is reasonably robust. If you would like to experiment with it, please contact one of the authors.

References

[1] Coplien, J. O., "Looking over one's shoulder at a C++ program" 11265-921030-01TM

[2] Devanbu, P., "GENOA/GENII - A customizable, language- and front-end- independent code analyzer", *Fourteenth International Conference on Software Engineering*, Melbourne, Australia, 1992.

[3] Grass J., and Chen, Y-H., "The C++ Information Abstractor", Proceedings, *The Second USENIX C++ Conference*, San Fransisco, USA, 1992.

[4] Murray, R.B., "Statically Typed Abstract Representation for C++ Programs", Second USENIX C++ Conference, 1992.