Reprinted from the

Proceedings of the GCC Developers' Summit

June 17th–19th, 2008
Ottawa, Ontario
Canada

Using GCC Instead of Grep and Sed

Taras Glek and David Mandelin
Mozilla Corporation
tglek@mozilla.com dmandelin@mozilla.com

Abstract

Large codebases benefit from automatic assistance when enforcing correct API usage and performing large-scale refactoring. To make large-scale refactoring of the Mozilla codebase successful, we developed a static analysis framework based on a time-tested GCC compiler infrastructure featuring the expressiveness and ease of prototyping of a general-purpose scripting language.

The presentation will describe our GCC plugin system and two generic static analysis plugins, Dehydra and Treehydra. Dehydra exposes a simplified view of the GCC AST, suitable for novice users and semantic-grep-style analyses. Treehydra exposes the GCC GIMPLE API. We also describe our static analysis applications and experience with the system.

1 Introduction

This paper describes Mozilla's experience using GCC as a front end for static analysis. We believe static analysis tools are required to assist with large-scale code changes planned for Mozilla 2, the next version of our browser platform. For example, developers are working to replace the reference-counting memory management subsystem with MMGC, a conservative garbage collector. This work requires allocation sites for thousands of classes to be changed according to various patterns, such as replacing reference counts with write barriers. We believe this project is much more likely to succeed with help from static analysis.

Refactoring needs were the deciding factor for delving into development of static analysis tools for Mozilla. However, static analysis is useful for everyday C++ development tasks. The creator of C++, Bjarne Stroustrup, describes C++ as a multi-paradigm language. It was not designed to encourage best practices for any specific styles of programming. Unfortunately, the language leaves it up to the user the ensure that only the correct paradigms are used. For a large programming project like Mozilla, this causes a significant review burden to ensure that all code adheres to the Mozilla C++ paradigm. Stroustrup even goes as far as to propose restricting C++ semantics through library-specific static analysis [21]. Perhaps a future C++ standard will mandate static analysis capabilities in compilers.

Our solution to this problem is a lightweight plugin system for GCC and three plugins: Dehydra, Treehydra, and PluginMultiplexor.

Dehydra and Treehydra are analysis plugins that expose different GCC intermediate representations to JavaScript. Thus, the plugins do not implement analyses directly. Rather, they form a system for scripting application-specific analyses in a high-level language.

Dehydra exposes a high-level simplified view of the AST, intended for simple analyses and "semantic grep" applications.

Treehydra exposes GCC's GIMPLE and is intended for traditional static analyses.

PluginMultiplexor is a utility plugin that runs multiple plugins in sequence within a single GCC invocation, improving total analysis run time and demonstrating the flexibility of the plugin system.

Needs-based development model. Treehydra and Dehydra are different from most static analysis tools because they were developed to meet clear practical needs at Mozilla. We started out with a list of problems that required automation and then set out to solve those problems. As a result, the tools progressed quickly from concept to production use. This is uncommon in the static analysis field. More typically, experimenters create a tool to try out an interesting idea or technique, verify it on small problems, then make attempts to find practical uses for the tool and scale the tool to real software.

Our static analysis needs fall into two main classes: verifying that coding conventions are followed, and driving cross-cutting source rewrites. The MMGC refactoring provides an example of both. In this refactoring, `nsCOMPtr`, originally a smart pointer template class, will become a template class used to add write barriers[1] to class member fields. This change makes existing classes use write barriers "automatically" with no extra work. But smart pointers are also used on the stack, and it makes no sense to construct write barrier wrapper class object on the stack. So there is a new coding rule: an object of a write barrier type cannot be declared on the stack. There is also a rewriting task: replace all stack declarations `nsCOMPtr<T> x` with `T x`. We have used Dehydra to automate both the checking and rewriting.

We are also discovering applications for Dehydra in understanding our large code base. Dehydra is being used to extract class hierarchies, build call graphs, and identify classes that need further manual attention. We hope the wealth of information easily accessible through Dehydra will be used to build an improved, more semantically aware version of the LXR/MXR [10] online code browser.

Analyzing C++

A requirement for for Mozilla is analyzing C++. In a typical Linux Firefox build, GCC compiles 1906 C++ files and 421 C files. Thus C++ is the most important language for us to analyze. This contrasts with most existing analysis projects, which have started with C or Java because those languages are simpler. Even if the analysis technique extends to C++, the creators typically do not implement their technique in C++, because the research payoff is low compared to the difficulty.

Perhaps even harder than analyzing C++ is the first step: parsing C++. There are ambiguities in the parse tree that can be resolved only by semantic analysis, which in turn must correctly implement intricate rules for overloading, inheritance, and templates, and apply them correctly to an ambiguous parse tree. Thus, a static analysis tool creator must use an existing C++ parser in order to complete the project in less than several years. (See Section 2.5.1 for a comparison of existing parsers.)

But although C++ is our primary target, we believe our tool architecture generalizes cleanly to other GCC-supported languages. Supporting a new language simply requires applying our adapter and conversion code to the desired GCC front end.

[1]A write barrier is an action to be taken before a memory location is written, and is used to synchronize application code with a garbage collector.

2 Analysis Framework Design

2.1 Motivation: Application-Specific Analyses

There have been many successful general-purpose static analyses. The most successful drive compiler optimizations are used every day. General-purpose verifiers, such as compiler checks that generate warnings, are also successful. Popular general-purpose analyses tend to be coded right into the compiler so everyone can use them easily.

Mozilla (as well as others, we believe) also has application-specific and project-specific analysis needs, such as our MMGC garbage collection refactoring example.

MMGC requires a checker for a second application-specific rule, *finalizer safety*: GC finalizers (destructors, in MMGC) are not allowed to dereference member field pointers, because the pointed-to objects may already have been collected. This includes the case where the finalizer calls another method that dereferences a member.

Another application-specific rule is our protocol for outparams (params used to pass values out of a function): the outparams must be written to if and only if the function returns a success code. Note that there seems to be no good way to enforce this rule within C++: it requires an external analysis. (C++ provides type checking, but the rule depends on value and control flow properties.)

Manually enforcing all these rules on a large code base ($\sim$3MLOC C++) is intractable: we really do need automatic checkers. And we would like the checkers to be part of the compiler, so violations can provide developers with errors and warning messages, just as usual.

One way to produce such compiler errors is by expressing checks in the language, by static assertions or clever use of the type system. Mozilla does use that technique, but it does not apply to our examples here. For the write-barrier example, C++ provides no way to constrain what types can be constructed on the stack. The other examples depend on properties of the call graph, control flow, and value flow, which cannot be checked in the type system of C++.

Thus, we require a system that can (a) express analyses that are too complex to express inside C++, and (b) integrate the analyses with the compiler, but (c) does not require hard-coding the analyses into the compiler. Compiler plugins, modules that are loaded dynamically by the compiler, are a natural way to meet these requirements.

2.2 Scripting Languages for Program Analysis

The next major design issue is the language used to express application-specific analyses. In this section, we will argue that dynamic languages are an excellent choice.

The designer of a program analysis system must select a language used for programming the application analyses. We can describe this design space in terms of four broad language groups: (a) compiler implementation languages, (b) logic languages, (c) query languages, and (d) scripting languages.

Compiler implementation languages. Analyses can be coded up like any other part of the compiler, which is typically written in a typed compiled language (e.g., C, ML, Java). Then the analysis framework is simply an API exposing compiler data structures, and possibly libraries for program analysis. The advantages of this approach are that the language is familiar to compiler writers, that all compiler information is exposed, and that typed languages run fast. Examples of this approach include sparse [22], oink [18], and CLL [1].

Logic languages. Many research systems allow analyses to be specified declaratively as logical formulas or constraints, in languages such as Datalog or first-order or other logics. Logic languages often yield very concise analyses. An important additional benefit is that because the programs are short and free of engineering details (data structure choices and solver designs), prototyping analyses is fast and easy. But these languages are hard to make run as fast as hand-tuned analyses, and more importantly for us, these languages are easily used and understood only by specialists. As a result, usage of systems based on this approach tend to be limited to academic research. Examples of this approach include Saturn [9], Stratego [12], and bddbddb [24].

Query Languages. The main problem with logic languages is that they are unfamilar and intimidating, so some systems have designed analysis languages more in the spirit of SQL. For example, Mygcc allows users to specify sequences of calls that are allowed on execution paths. These languages allow concise analyses and easy prototyping, and they are also easy to use for nonspecialists. The disadvantage is that the language is limited to a certain class of analysis. Another example of this approach is UNO [18].

Scripting languages. A final choice is to write analyses in a "scripting language," i.e., a very high-level dynamic language, such as Python, or a query language. This idea is similar to using a compiler implementation language, in that the analysis system can potentially expose all compiler data, the language is familiar, and the language is general-purpose, so it can express any analysis. But dynamic languages can also be made to look like other, domain-specific languages relatively easily, so the same system can support query languages as a library. Finally, although the analysis algorithms must be written in the dynamic language, many data structure and memory management details are hidden from the user, and there is no compiler type checker to satisfy before running a test, so prototyping is relatively easy.

One risk of this approach, compared to using a compiler implementation language, is that code must be written and maintained to reflect the intermediate representation or some subset of it as scripting-language objects. Thus, success hinges on being able to conveniently implement data conversions for complex compiler data structures.

Because the analysis is run under an interpreter, another risk is that runtime performance will be unacceptably slow. However, modern scripting language interpreters are competitive with ahead-of-time compilation for many workloads, and we can always implement performance-sensitive analysis kernels in the compiler implementation language. Also, analysis of a large code base is "embarrassingly parallelizable," just like compilation itself.

We are not aware of any other tools that provide scriptable static analysis via bindings to high-level, general-purpose languages.

```
var classes = []
function process_type (c) {
  if (/class|struct/(c.kind))
    classes.push (c.name)
}

function input_end() {
  print (classes.length + `` classes instantiated'')
}
```

Figure 1: Dehydra script to count class instatiations

2.3 Scripting with JavaScript

We wanted Mozilla developers, most of whom are not compiler specialists, to be able to write their own application-specific analyses with minimal effort, so we decided to write analyses in a scripting language.

Specifically, we chose JavaScript, because it is a high-level, concise dynamic language with C-like syntax that is well known by Mozilla developers. The same benefits could be realized with similar dynamic languages such as Python or Scheme. If there is sufficient demand outside of Mozilla, it is likely that a future version of our plugins will provide a common API to support multiple scripting languages.

Our experience shows that the expected benefits of scripting languages hold up in practice. Figure 1 shows that an example of a concise JavaScript analysis. Non-specialists can use Dehydra—the MMGC work is being done by a non-compiler-specialist. But Treehydra is general enough to support classic analyses like live variable analysis, as well as related application-specific analyses, such as the outparams example.

2.4 Scaling to Mozilla with GCC

There are some special issues to be addressed in running an analysis in practice on a large code base like Mozilla. First, the parser must be able to parse all the C++ constructs used. To date, Mozilla makes little use of newer C++ features like partial template specialization, but these features are being used more and more, and they are starting to appear in standard header files, so we need a high-quality, up-to-date parser. Second, because of the size of the code base, the analysis must support separate compilation and latter combination of analysis results. Third, the analysis system should be easy to deploy (e.g., not require downloading and building obscure parsing and analysis packages). Finally, the tool should integrate easily with the build system. It is difficult to replay the right preprocessor definitions and related compilation options (e.g., `-fshort-wchar`) on every file outside of the build—build integration is the clear practical solution.

All these requirements are met by GCC as a front end and point of invocation of the analysis scripts.

2.5 Related Work

2.5.1 Choosing a C++ Parser

One can't write a good static analysis tool without a high-quality parser.

EDG [17] CFront appears to be commonly used by researchers, but its proprietary nature makes it inappropriate for Mozilla.

LLVM's Clang [2] project is very promising for static analysis needs: it is being designed from the ground up as a set of reusable components with static analysis and refactoring in mind. However, Clang will not support C++ in the near future.

Elsa [20] is a C++ front end designed around the Elkhound [19] GLR [5] parser, Elkhound [19]. Elsa is clearly laid out. In particular, the GLR system makes extending the grammar easy. Unfortunately, Elsa does not have an active community and it is not actively maintained, and it is not able to completely parse modern C++. Older versions of Elsa could parse older versions of Mozilla on older versions of GCC headers. But even after additional work by Mozilla, on current Mozilla and

GCC headers, Elsa cannot parse one file, and has incomplete support for some C++ features, such as implicit type conversions in AST.

Before we discovered Elsa's limitations, we did not consider using GCC, because GCC has a negative reputation for a hard-to-use API and for political problems when reusing the front end or adding plugin support. But once we decided that maintaining a complete C++ front end by ourselves was not plausible, we tried using GCC. We were pleasantly surprised how easy it was to hook into GCC for plugins. We found the API difficult at first, but learnable, and it translates cleanly to JavaScript.

2.5.2 Analysis Frameworks

Oink Suite. The Oink suite [25] consists of the Elsa C/C++ parser, CQual++, a type qualifier analysis implemented using Elsa, and some other special-purpose Elsa-based analysis tools. The tools are implemented in a simple framework for making Elsa-based applications, and users are expected to reuse this framework for their own tools.

Oink includes libraries for tracking interprocedural value flows, used to drive the type qualifier analysis, but has no facility for traditional CFG intermediate-representation-based data-flows analysis.

UNO. UNO [18] is a static analysis tool for C with a domain-specific language for checking user-defined properties. UNO features a simple imperative DSL with a C-like syntax. The DSL allows one to pattern match paths in a control flow graph in order to find incorrect paths. The DSL does not have a concept of types and is limited to intraprocedural analysis.

Unfortunately, UNO is based on a primitive C parser which does not support all C code present in Mozilla, or parse C++. Technical limitations aside, UNO is easy to learn and use, the simple DSL has room for future extensibility. The DSL is simple and demonstrates that detecting certain classes of control-flow senstive patterns can be simple.

Mygcc. Mygcc [23] is a static analysis tool for C that integrates into GCC, so it can handle any C that GCC can, and it can be run as part of a build.

Mygcc is similar to UNO in that it is designed for intraprocedural analysis. It also does pattern matching over a control flow graph, but unlike UNO, it uses a more restrictive declarative DSL. Like UNO, it also does not pattern-match on types. It is unclear if it supports C++ or if the DSL could be extended to allow user analyses other than path-based error checking.

2.5.3 Object Wrapping

In C++ one can wrap primitives in objects in order to restrict their usage. While this is a technique rather than a tool, it deserves a mention as it the most widespread error-checking technique used in place of separate static analysis. For example, in Mozilla developers frequently attempt to restrict usage of certain constructs by wrapping them in templates or plain classes. However, it is not possible to express everything with C++ classes and it can lead to an increase in compilation time, increase in code size, and decreased performance.

Tracking Units. There are a lot units used in Mozilla layout code: twips, screen pixels, CSS pixels, millimeters, etc. [7]. An attempt was made to enforce correct conversions between these types by wrapping variables used to represent them in classes. However this resulted in slower code [13]. A static analysis solution would make use of typedefs to extract unit information without interfering with the quality of compiled code.

3 Dehydra

3.1 Introduction

Dehydra[2] is the flagship GCC plugin developed at Mozilla. Dehydra provides a callback-based JavaScript API in place of a custom DSL as is often the case in static analysis tools. With Dehydra, one can even write useful scripts in 3-4 lines of JavaScript—Dehydra can realistically be used as a quick and dirty, semantically-aware grep.

[2]The original version of Dehydra provided control flow information (this functionality is now provided by Treehydra) which, when graphed, makes functions vaguely resemble the multi-headed Hydra monster from Greek mythology. Iterating over the CFG to solve bugs can be thought of as decapitating the mythical Hydra monster.

As shown by UNO, a simplified subset of the AST is enough for some useful analyses, and the resulting scripts can be easier to write. The omissions restrict the analysis domain, but carefully selecting the omissions can lead to a powerful, yet simple, static analysis system. The idea of Dehydra is to provide a minimal subset of the AST that permits useful analyses. Each Dehydra feature was added to meet a specific Mozilla analysis need. As a result, the Dehydra API serves as a summary of basic static analysis needs.

Most existing Dehydra applications exclusively analyze type declarations. Thus Dehydra presents detailed information about AST types. On the other hand, having complete function bodies turned out to be much less important. Dehydra presents function bodies as a list of assignments and function calls, including arguments. For more information on Dehydra data structures, refer to the Dehydra documentation [3].

The large gap in Dehydra features for querying the type system and function bodies can be explained by the fact that the C++ inheritance hierarchy is often very large. Due to template instantiation and inheritance, it is harder to follow by visual inspection than a function body. Additionally, function bodies are much more verbose and the analysis effort is greater for finding patterns in AST structures than in type hierarchies.

To address analysis needs of scripts where the AST exposed by Dehydra is insufficient and it's unreasonable to extend the Dehydra AST, Treehydra was conceived.

3.2 API

Dehydra feeds AST information to use scripts via the following callbacks:

`process_type(`*`t`*`)` This is called for each complete declaration of an aggregate type. `t` is a JavaScript object describing the type.

`process_function(`*`f`*`, `*`body`*`)` This is called for each function definition. `f` is the declaration. `body` is a flattened list of statements that contains only assignments, references to fields and variables, and function calls. Control flow and arithmetic operators are omitted.

`process_var(`*`v`*`)` This is called for all global variable declarations. `v` is the Dehydra representation of the variable.

`input_end()` This is called once at the end of compilation. It can be used to finish or summarize analyses.

3.3 Current Status

Dehydra has been implemented for GCC 4.3 on Linux, and subsequently ported to MacOS GCC 4.2.

Dehydra has been integrated into the build system in the main Mozilla 2 source code repository, and several Dehydra-based checks can be run with a standard build.

Existing applications include:

- Visualization of an inheritance hierarchy.
- Enforcement of gargage-collection safety.
- Finding struct packing bugs.
- In combination with sed for semantically-aware refactoring: Find all classes that inherit from nsISupports and pass that list to a script.
- Various API misuse bugs.

4 Treehydra

4.1 Introduction

Treehydra[3] is a GCC plugin that exposes the GCC GIMPLE data structures as JavaScript objects to enable low-level analyses. The mapping is direct: an instance of a GCC struct or union is represented by a JavaScript object with the same fields.

Treehydra was created because certain analyses need precise information about program AST and control flow that is not available in Dehydra's minimal AST. For example, unit conversion checking (see Section 2.5.3) requires a view of all arithmetic operators. Another example is the outparam check (see Section 4.5), which requires control flow and comparison operators.

We have found GIMPLE to be a good intermediate representation for analysis, although not without disadvantages. Compared to the AST, GIMPLE much more

[3]The "Tree" part of the name refers to the GIMPLE tree typedef around which the plugin is built.

clearly exposes the semantics of the program, and has fewer node types for analyses to handle. But we expect mapping GIMPLE code back to the original code (e.g., to rewrite code constructs corresponding to a certain GIMPLE basic block) to be difficult. Also, some optimization occurs before CFG generation, which is undesirable for analysis.

Another possible IR is GIMPLE-SSA, but we do not expect SSA form would simply our applications much. Therefore, considering that even more optimization takes place before SSA generation, and the cost of learning the apparently more complex SSA APIs, we have not tried SSA.

4.2 Implementation

Treehydra inserts a new analysis pass into GCC at a user-specified point, usually right after CFG generation. It then calls the user script with `current_function_decl` for every function that is compiled.

Conversion of GCC trees to JavaScript object is performed by C code generated automatically from GCC header files using Dehydra.

GCC structs use gengtype GTY annotations to represent additional information about array lengths and union tags, and we wanted to access this data in our conversion generator. Doing so in Dehydra required us to convert the GTY annotations into standard GCC annotations. This is somewhat unfortunate because it required modifying gengtype and moving attributes a bit in most header files in order to conform with standard attribute syntax. For example, `struct name GTY(())` became `struct GTY(()) name`.

The first prototype of Treehydra converted the entire `current_function_decl` global variable in a depth-first traversal on every struct and union member, but the current version converts structs on demand. We found that converting the entire tree was slow, and usually most of the data was never accessed. Now Treehydra builds the trees lazily as they are accessed.

A pointer map is used to cache pointers to C structs, mapping them to their JavaScript representation. Each GCC struct instance is represented by a single JavaScript object. This ensures that if two pointers are equal in C, the same equality applies in JavaScript. The map is reset on every plugin pass[4] to avoid discrepancies. This, combined with the fact that Treeydra uses lazy tree nodes, means that Treehydra scripts are not allowed to reference tree nodes beyond the lifetime of the `process_tree` function.

The reflection strategy used in Treehydra could be made to work both ways. In theory, one could prototype optimization passes in JavaScript. However, Treehydra is meant as a static analysis tool—the data structions are provided for inspection only.

4.3 API

Treehydra calls the user JavaScript function `process_tree(current_function_decl)` for each C++ function. As mentioned in 4.1, the JavaScript objects are isomorphic to GCC structs and unions. Treehydra also includes a standard JS library with functions parallel to GCC macros and basic access functions, including `TREE_CODE`, `DECL_NAME`, and `walk_tree`. Porting this code to JavaScript is simple because of JavaScript's C-like syntax and Treehydra's GCC-like representations.

JavaScript's dynamic nature provides opportunities to simplify the API. For example, by adding a property accessor to the JavaScript prototype used for GCC objects, the user can type `decl.name` instead of `DECL_NAME(decl)`. JavaScript also makes it possible to dump object representations, and even search object graphs for desired information, making it easier to learn and understand the API.

4.4 Comparison with Dehydra

Treehydra is much more suitable than Dehydra for analyses that require detailed analysis of function body semantics.

Most of the Treehydra's C code is generated by a 500-line JavaScript program, so Treehydra is easier to maintain. Treehydra also hooks into GCC at fewer points. Note that the code generator is run using Dehydra, although Treehydra needs only a subset of what Dehydra provides. In fact it would be possible to use Treehydra to boostrap Treehydra since the conversion process relies purely on types which are maintained in GIMPLE.

[4] In GCC, each optimization pass is invoked once per function being compiled.

However, by relying on a separate tool, bootstrap issues are avoided.

The Dehydra API is simpler, easier to learn, and fits better with JavaScript. The Treehydra API is a good fit with JavaScript, but Dehydra is even better. For example, the generic JavaScript print function works reasonably well with Dehydra objects, but crashes on Treehydra objects because it ends up trying to assemble a printout of the entire GCC object graph at once and runs out of space.

Dehydra also has a few advantages in semantics because it uses an AST from before gimplification. For example, consider an analysis to detect code where a variable is both used and modified between two sequence points, so the order of use and modification is undefined by the language. The notion of sequence points exists only in the AST, so it is not possible to detect this error by analyzing GIMPLE in Treehydra.

4.5 Current Status

As of this writing, Treehydra runs, but does not yet provide complete access to the GCC's tree data structure. Some tagged unions are not yet reflected. For example, nodes representing template information are missing. However, Treehydra is already useful, as shown by the examples below, which are nearing completion.

Finalizer Safety [4] In a garbage-collected system, finalizers (destructors, in MMGC) should not dereference pointers to garbage-collected objects, because the pointer may be dangling by the time the finalizer is called.

We define a class method as *finalizer-safe* if it does not dereference pointers to garbage-collected classes. Note that a finalizer-safe method can only call other finalizer-safe methods, so the property is transitive.

We are writing an analysis to verify finalizer-safety of destructors. The existing version relies on methods being manually annotated (using GCC attributes) as finalizer-safe. In the future, the analysis will also analyze all the deference operations.

Rootedness [6]. SpiderMonkey [11] utilizes a precise garbage collector that does not scan the C stack. Thus, when a native function holds a pointer to a JavaScript object, it must either store a pointer to the object in a property of a rooted object, or specially register the object with the interpreter as an additional GC root before calling any interpreter API method that may invoke GC. A Treehydra analysis is being prototyped to check this property.

Outparam Analysis [8]. Treehydra has been used to implement an analysis to check that outparams (pointer or reference function parameters used to return arguments) are used according to the Mozilla convention. In this section, we describe a simplified version of the problem, leaving out unimportant practical details.

The rule to be enforced is that if the function returns zero (indicating success), all outparams must have been modified, and if the function returns nonzero (indicating failure), no outparams may have been modified.

The Treehydra script implements an ESP [15]-style abstract interpretation to track which outparams have been modified and the values of all variables that can reach a return statement. The analysis totals about 2000 lines of JavaScript, including GCC API adapters, data structures for analysis, a basic abstract interpretation framework, the ESP framework, a live-variables analysis, a return value reachability analysis, and the outparam analysis itself. Only 400 lines of code are specific to this analysis.

Currently, the analysis runs correctly on a test suite of small programs. Improving it to run on Mozilla is ongoing work, mostly involving adding cases to the analysis to handle all the GCC GIMPLE tree node types used. We know that the analysis will scale to Mozilla because we have successfully run a Python prototype using a cruder access method to get at GIMPLE.

5 GCC Strengths and Limitations for Static Analysis

5.1 Strengths

The main advantage of GCC is that developers already have and use GCC. Thus, there is no compiler framework to deploy, and analysis applications can be easily integrated with build systems. Also, almost all code compiles with GCC, so GCC is an almost perfectly compatible front end.

GCC attributes provide a standardized, parser-friendly, built-in facility for application-specific annotations. Other static analysis systems are often forced to use ad-hoc approaches such as embedding annotations into comments or customizing the parser.

GCC retains most type information, including typedefs. This is important because source code type names often have application-specific semantics which are to be checked by analyses. For example, the Mozilla types `PRBool` and `nsresult` are both integers, but we consider code that assigns values of one type to the other to be incorrect.

The vibrant GCC developer community made it easy to get started with what would otherwise be an intimidating codebase. The `#gcc` IRC channel and the GCC mailing list were helpful resources to help guide us through the codebase. This is the opposite of the experience we had developing with Elsa, which lacks an active user community, making reading the source the primary way of learning.

The dynamically typed nature of tree nodes was easy and natural to reflect into JavaScript's dynamic type system.

The GIMPLE representation proved to be straightforward to work with.

Cross-compiler configurations allow checking code on multiple platforms (essential for Mozilla) without having to port the static analysis tool to those platforms. Most static analysis tools are not portable beyond their primary platform and don't support cross-compilation.

Finally, GCC has excellent parsing performance compared to the alternatives used by Mozilla. Both Elsa and the MCPP preprocessor we use with Elsa are significantly slower than GCC.

In conclusion, GCC compatibility, internal design, performance, developer community, and user base make GCC an attractive base for building a static analysis tool.

5.2 Limitations

Early optimization, such as early constant folding, make GCC's AST diverge from the input code, and make some analysis more difficult or impossible, e.g., a style check that variables are initialized at declaration is impossible.

Some typedef information is dropped as an optimization, such as typedefs declared within class bodies. It appears this is done to simplify error messages for template instantiations. A better solution for analysis would be to preserve typedefs and chase them as necessary when generating error messages.

Early gimplification makes it hard to access non-gimplified function bodies during compilation, requiring more plugin hooks in the front end. This in turn makes it harder to implement the system in additional front ends. Informal conversations suggest that GCC on the compile-server branch keeps non-gimplified ASTs, which would solve this problem.

In-place tree mutation is frustrating because trees cannot be saved for later analysis. Dehydra copes by converting trees to JavaScript early and then saving them until all type information is available. Treehydra is forced to limit object lifetimes to a single plugin callback.

Node reuse for `DECL`s and other node types makes it hard to find position information in function bodies. Unfortunately, this combined with early optimizations such as constant folding, precluded using GCC to rewrite source code. For that reason Mozilla continues to use Elsa for most automated rewritings.

The C++ front end has excellent `*_as_string(tree, flags)` functions. However, they do not work consistently due to calling into C code which disregards the `flags` parameter, which reduces the utility of these functions.

Other than tree node reuse and in-place tree modification, none of these limitations would be hard to overcome, thus the benefits of plugging into the GCC codebase greatly outweigh limitations.

5.3 Plugging Into GCC

The key advantage of plugins is easy deployment of application-specific analyses, but we noticed that plugins also add flexibility, allowing us to use libraries that don't belong in a compiler, such as the Spidermonkey JavaScript interpreter.

The plugin implementation, especially the API used by Treehydra, is similar to that described in "Extending GCC with Modular GIMPLE Optimizations" [14] at the 2007 summit.

GCC was modified to link with `-rdynamic -ldl` in order to load plugins and allow them to access GCC data structures. Surprisingly few modifications were needed to implement the plugin interface for static analysis needs. Currently the patch weighs in at 14 kilobytes.

In order to load plugins, two command-line switches were added to GCC:

`-fplugin-name=` indicates the dynamic library to load.

`-fplugin-arg=` is used to specify the command line to pass to the plugin. In Dehydra and Treehydra, it indicates the script filenames to load.

The plugin inteface consists of the following callbacks:

`gcc_plugin_init` initializes the plugin. Its arguments are the pathname of the plugin being initialized, and a string passed on the command line via the `-fplugin-arg=` option. It may report the name of an optimization pass (using the names in `passes.c`); if so, `gcc_plugin_pass` will be called after that pass is complete.

`gcc_plugin_cp_pre_genericize` is called for each C++ function declaration, and is provided with the parse tree of that function before lowering to GENERIC. Some optimizations, unfortunately, have already been done at this point (see Section 5.2).

`gcc_plugin_finish_struct` is called for each complete `struct` declaration.

`gcc_plugin_post_parse` is called after an entire translation unit has been parsed, while the GCC C++ front end variable `global_namespace` is still available for iteration.

`gcc_plugin_pass` is called as if it were an optimization pass, immediately after the pass named by `gcc_plugin_init`.

`gcc_plugin_finish` is called at the end of compilation.

The above functions provide the minimal functionality needed in order to provide static analysis capabilities in plugins. The rest of the plugin API consists of the internal GCC API. Note, we are against hindering GCC development by requiring a stable plugin interface. Instead, the burden is shifted to plugin authors. The plugin API is intentionally minimal to minimize the amount of effort required to port plugins to future versions of GCC. The plugins are meant to be adapters that provides a consistent API to plugin users. So far this strategy has worked well the plugins described in this paper.

5.4 Benefits to GCC

We believe that are benefits to be reaped from our work for both GCC users and GCC developers. GCC users get a powerful new way to ensure a more consistent, higher-quality codebase through static analysis. GCC developers should end up with a larger community since writing scripts and adding new features to the GCC plugins will induce more people to learn about GCC and provide motivation for further improvements to the GCC core.

Additionally, providing static analysis capability in GCC will give people yet another reason to make sure that they use GCC, which can only be a good thing.

6 Ongoing Work

Dehydra and Treehydra are currently under active development. Dehydra is almost ready for a 1.0 release. Treehydra is being worked on to reflect GIMPLE more completely.

The PluginMultiplexor plugin is not yet completed.

Additionally, work is being done to make using GIMPLE under Treehydra as intuitive as possible by exploiting JavaScript prototypes and making objects more self-documenting.

We are looking forward to ongoing improvements in GCC such as better location tracking and improved performance. In particular, the compile server branch of GCC could prove to be a big performance boost for our static analysis system. As we wrap up work on the current GCC plugins, we intend to start contributing improvements back to GCC. Ideally we will start addressing some of the limitations discussed in Section 5.2.

7 Performance

Performance has not been a foremost concern during development of Dehydra and Treehydra. From casual observations, it appears that Dehydra causes a measurable but not excessive compilation slowdown.

Currently a debug Mozilla 2 build takes 27 minutes to build without loading GCC plugins. With Dehydra and a barebone script, the build takes 30 minutes. A barebone Treehydra script makes the difference even smaller due to Treehydra's lazy nature. In practice, the static analysis performance penalty depends on the analysis script.

8 Conclusion

In this paper, we have argued for the need for static analysis tools for checking, understanding, and refactoring large codebases. Further, we argued that the tools should make it easy for both experts and novices to design, prototype, and test application-specific analysis tools, and implement analyses. We then described our tool chain that realizes this goal: GCC plugins, which allow the use of GCC in a prototyping setting; Dehydra, a plugin that exposes GCC ASTs to simple JavaScript analyses; and Treehydra, another JavaScript-based plugin that exposes GCC GIMPLE representations to advanced JavaScript analyses. This tool chain allows users to write programs in a familiar, general-purpose dynamic language that analyze program representations in a standards-compliant multi-language production compiler.

The tool chain is new, but already has led to improvements in production Mozilla code, and we are still finding and creating new analysis applications. Some of these analyses were implemented by compiler novices using Dehydra. On the other side, we have demonstrated that Treehydra can support analyses just as complex as those required for compiler optimizations. Both plugins are needed: the intermediate representations and GCC internals API exposed by Treehydra are too difficult for compiler novices, but the power of Treehydra is required for advanced applications.

Downloads and some usage information on the tools described are available on the Dehydra homepage [16].

References

[1] CIL (C Intermediate Language). `http://cil.sf.net`.

[2] clang: a C language family frontend for LLVM. `http://clang.llvm.org/`.

[3] Dehydra API. `http://wiki.mozilla.org/Dehydra_API`.

[4] Finalizer safety. `https://bugzilla.mozilla.org/show_bug.cgi?id=424416`.

[5] GLR algorithm. `http://en.wikipedia.org/wiki/GLR_parser`.

[6] Implement GC-safety analysis for SpiderMonkey. `https://bugzilla.mozilla.org/show_bug.cgi?id=421934`.

[7] List of Units in Mozilla Layout. `http://wiki.mozilla.org/Mozilla2:Units`.

[8] Outparam analysis. `https://bugzilla.mozilla.org/show_bug.cgi?id=420933`.

[9] Precise and scalable software analysis. `http://saturn.stanford.edu/`.

[10] Software toolset for indexing and presenting source code repositories. `http://lxr.linux.no/`.

[11] SpiderMonkey: JavaScript Engine. `http://www.mozilla.org/js/spidermonkey/`.

[12] Stratego a language and toolset for program transformation. `http://strategoxt.org/`.

[13] Unit conversion. `https://bugzilla.mozilla.org/show_bug.cgi?id=265084`.

[14] Sean Callanan, Daniel Dean, and Erez Zadok. Extending GCC with Modular GIMPLE Optimizations. Ottawa, ON, Canada, 2007. GCC Summit.

[15] Manuvir Das, Sorin Lerner, and Mark Seigle. ESP: path-sensitive program verification in polynomial time.

[16] Taras Glek. Dehydra Wiki. `http://wiki.mozilla.org/Dehydra_GCC`.

[17] Edison Design Group. EDG C++ Frontend, 2008. `http://www.edg.com/index.php?location=c_frontend`.

[18] Gerard J. Holzmann. UNO, 2008. `http://www.spinroot.com/uno/`.

[19] Scott McPeak. Elkhound parser, 2008. `http://www.cs.berkeley.edu/~smcpeak/elkhound/`.

[20] Scott McPeak. Elsa C++ Frontend, 2008. `http://www.cs.berkeley.edu/~smcpeak/elkhound/sources/elsa/`.

[21] Bjarne Stroustrup and Gabriel Dos Reis. The pivot – a brief overview. `http://charm.cs.uiuc.edu/patHPC/slides/stroustrup-a.pdf`.

[22] Linus Torvalds. Sparse: static analysis tool for linux kernel. `http://www.kernel.org/pub/software/devel/sparse/`.

[23] Nic Volanschi and Sebastian Pop. Mygcc, 2008. `http://mygcc.free.fr`.

[24] John Whaley. BDD-Based Deductive DataBase. `http://bddbddb.sourceforge.net/`.

[25] Daniel S. Wilkerson, Karl Chen, and Scott McPeak. Oink Suite, 2008. `http://www.cubewano.org/oink`.