Automatically Transforming GNU C Source Code

Christopher Dahn, Spiros Mancoridis
Department of Computer Science
Drexel University, Philadelphia, PA, USA
{chris.dahn}@computer.org
{mancors}@drexel.edu

Abstract

To perform automated transformation techniques on production quality GNU C source code, non-trivial normalizations must occur. The syntax of GNU C contains inherent ambiguity that must be overcome. The techniques used by an automated transformation tool, Gemini, are presented.

1 Introduction

The process of performing automated source code transformation on GNU C code is complicated. GNU C is a flexible language that allows developers to write code in a variety of ways that are all semantically equivalent. This flexibility imposes challenges to software maintainers who wish to perform source transformation automatically.

In this paper we look at techniques used by the Gemini [4, 3] tool, developed at Drexel University, which performs source code transformation, automatically, on GNU C source code. The tool transforms GNU C arrays into pointers. The transformation must take into account the presence of non-C input (e.g., preprocessor statements) and an ambiguous language definition, while maintaining semantic equivalence of the input and output source code.

Gemini uses the GNU C Compiler (GCC) to perform preprocessing of GNU C source code, and TXL [1, 2, 5] to perform the automated transformation.

The remainder of this paper is structured as follows: Section 2 discusses some of the specific challenges posed by GNU C, Section 3 discusses the details of the TXL transformations used to normalize the source code, and Section 4 discusses the conclusions we can draw from the study.

2 Challenges of GNU C

2.1 Preprocessor Input

Gemini cannot transform GNU C source code that contains macros, since macros are resolved at compile-time. Therefore, Gemini passes the source code through the GCC preprocessor before transformation.

Gemini's preprocessing performs two primary tasks. First, it pre-processes the original source code with the GCC preprocessor. Second, it removes all of the preprocessor statements from the result.

The output from the GCC preprocessor contains preprocessor statements which must be removed. These statements are used by GCC to reconstruct the original file scopes to resolve duplicate global declarations. It isn't possible to define a grammar that includes GNU C preprocessor statements everywhere they may occur. Generally, these statements can occur anywhere in the input. Creating an ambiguous grammar to describe them leads to parser generation problems.

2.2 C Declaration Ambiguity

GNU C is a flexible language. Its flexibility results in a complicated grammar. One of the complications found in GNU C grammars is embodied by an ambiguity in how declarations and statements are parsed.

In C, users can alias types using a `typedef` declaration. The presence of user-defined types causes GNU C grammars to be ambiguous about how to parse statements and declarations. Figure 1 and Figure 2 show possible grammars for a GNU C declaration and statement, respectively.

An example of a declaration described by the first grammar is, `my_type (foo);`. The statement grammar describes function calls with one argument, such as, `printf (foo);`. In the example declaration, the declarator type, `my_type`, is a user defined type alias (i.e., previously defined with a `typedef`).

```
<declaration> ::= <type> <declarator> ';'
<type> ::= <nativetype> | <typealias>
<nativetype> ::= int | char | void | float | double
<typealias> ::= <identifier>
<declarator> ::= <identifier> | '(' <identifier> ')'
```

Figure 1. BNF for one possible declaration grammar.

```
<statement> ::= <expression> ';'
<expression> ::= <postfixexpression>
                 <postfixextension>
<postfixexpression> ::= <identifier>
<postfixextension> ::= '(' <identifier> ')'
```

Figure 2. BNF for one possible statement grammar.

At the syntax level, both of these GNU C constructs have the same parse tree. A compiler would differentiate the former as a declaration and the latter as a statement during semantic analysis of the parse tree. However, TXL only operates on input syntax, hence the transformation that Gemini performs must first take steps to overcome this grammatical ambiguity.

3 TXL Transformation

The TXL transformation is composed of eleven steps. The relevant steps are outlined in this paper. Only one of the steps actually performs the array-to-buffer transformation, the other steps are required to normalize the way GNU C declarations are parsed (due to the flexibility of GNU C syntax), to account for the declaration/statement ambiguity previously discussed, and to ease the final array-to-buffer transformation.

Figure 3 shows an example GNU C program. This program is used to illustrate the eleven transformation steps. The lines affected by the current transformation step will be highlighted.

Step 1: Expand Shorthand Types. The first step is to find all declarations in the GNU C program that have been declared using a shorthand form of the `int` type. GNU C allows the integer qualifiers `long`, `short`, `unsigned`, and `signed` to be used as the declarator type. In this case, the GNU C grammar used with TXL will cause it to parse the declaration without a type. Hence, the parse tree for the declaration will have only a declaration type qualifier. In order to normalize declaration types, all declarations with

```
#include <stdio.h>

typedef struct
{
    char c;
} (char_buf)[5], char_st;

int main(int argc, char **argv)
{
    char *test = "test";
    char_buf (buffer) =
    {
       [0].c='t', [1].c='e', [2].c='s',
       [3].c='t', [4].c='\0'
    };
    char_buf buffer2;
    struct st
    {
       unsigned i;
    } st = {42};
}
```

Figure 3. Sample source code to be transformed.

```
struct st
{
    unsigned int i;
} st = {42};
```

Figure 4. Step 1: Expand shorthand types.

only type qualifiers are expanded to have an explicit type of `int`, as shown in Figure 4.

Step 2: Name Anonymous Elaborated Types. Elaborated types, or enumerated types, are user-defined types. In C, an elaborated type is either a `struct`, `union`, or `enum`. Figure 6 shows the two ways that an elaborated type can be declared.

In Figure 6, the type of the variable `foo` is an anonymous elaborated type. These types can only be used for a single declaration. That is, no other declaration can have the same type as `foo`, even if the type definitions of both declarations are identical.

The declaration of `foo2` defines a named elaborated type. This form is often used when the programmer intends to declare other variables with the same type. A subsequent declaration cannot declare the body of the `struct` again, rather, it references the `struct` by name (e.g., `struct st bar;`). In this case, the types of `foo2` and `bar` are equivalent.

```
typedef struct txl_WasAnonElabType1
{
    char c;
} (char_buf)[5], char_st;
```

Figure 5. Step 2: Name anonymous elaborated types.

```
struct
{
    int i;
} foo;

struct st
{
    int i;
} foo2;
```

Figure 6. Declaration of elaborated type.

In this step, a unique name is assigned to every anonymous elaborated type. Each name is unique in the namespace of the file, as shown in Figure 5. That is, the generated name will not be found anywhere else in the file undergoing transformation. Assigning a unique name to an anonymous elaborated type does not change the semantics of the program, since a unique name guarantees that the type has not been used anywhere else.

Step 3: Expand Declarator Lists. After all anonymous elaborated types in the program have been named, lists of declarators can be expanded, as shown in Figure 7. This is performed after naming anonymous elaborated types so that the proper type is distributed to each expanded declarator. If the anonymous elaborated type was distributed, then none of the new declarators would be equivalent in type which could cause type mismatch errors at compile time.

Step 4: Remove Extraneous Parentheses. Removing unnecessary parentheses from declarators, as shown in Figure 8, allows the assumption that a valid (i.e., transformable) declaration will never contain parentheses.

Generally, any number of parentheses may be added around a declarator without changing the semantics of the declaration. However, each set of parentheses creates a unique parse tree for the declaration. In particular, adding extraneous parentheses to declarations is the cause of the grammar ambiguity described in Section 2.2.

Due to the ambiguity, it is not possible to determine automatically if TXL has erroneously parsed a statement as a declaration. Removing the parentheses from a function call will always produce a syntax error at compile time. Hence,

```
typedef struct txl_WasAnonElabType1
{
    char c;
} (char_buf)[5];
typedef struct txl_WasAnonElabType1
    char_st;
```

Figure 7. Step 3: Expand declarator lists.

```
typedef struct txl_WasAnonElabType1
{
    char c;
} char_buf[5];

int main(int argc, char **argv)
{
    char *test = "test";
    char_buf buffer =
    {
       [0].c='t', [1].c='e', [2].c='s',
       [3].c='t', [4].c='\0'
    };
```

Figure 8. Step 4: Remove extraneous parentheses.

this process must be conservative in its selection. Specifically, if a declaration's type is an identifier, then it is assumed to be a statement (i.e., erroneously parsed) and the parentheses are not removed. This conservative approach allows the transformation to normalize all `typedef` declarations for Step 6. Subsequent steps will allow us to perform this step again to remove remaining false negatives (i.e., variable declarations that should have matched but were skipped).

Step 5: Unique Local Elaborated Types. This step, shown in Figure 9, ensures that when Step 6 removes all `typedef` aliases, local declarations will not have types that alias a globally defined elaborated type. This can lead to errors, since `typedef` flattening will invalidate scope protection of type declarators.

To make the types unique, the identifier of each elaborated type defined in the body of a function is replaced with a new, unique identifier. This takes into account forward declarations of elaborated types. Once a unique identifier has been chosen, all references to the previous type are replaced with references to the new, unique type.

Step 6: Flatten `typedef` aliases. This step is critical in determining which declarations to transform. This step also resolves any of the remaining ambiguities described in Section 2.2 and Step 4. The ambiguity is resolved since this

```
struct st1
{
    unsigned int i;
} st = {42};
```

Figure 9. Step 5: Unique local elaborated types.

```
struct txl_WasAnonElabType1 buffer[5] =
{
    [0].c='t', [1].c='e', [2].c='s',
    [3].c='t', [4].c='\0'
};
struct txl_WasAnonElabType1 buffer2[5];
```

Figure 10. Step 6: Flatten `typedef` aliases.

step replaces all identifiers that are type aliases with the type that they alias. Hence, all type aliases are resolved to the native or elaborated types they alias, as shown in Figure 10.

To flatten the `typedef` aliases, each `typedef` is visited once, starting at the top of the file being transformed. It can be assumed that the very first `typedef` in the file creates an alias to a native or elaborated type since it would be a compilation error to declare a `typedef` for an alias that has not been defined yet. For each `typedef` that is visited, every declaration below that `typedef` is visited to determine if the type of the declarator is the current `typedef` alias. If a match is found, the type alias of the declarator is replaced. This will also flatten `typedefs` themselves. For example, if a `typedef` defines an alias for a previously defined alias (e.g., `typedef my_char my_char2;`, where `my_char` is an alias for `char`), then it will be transformed to alias the original type (e.g., `typedef char my_char2;`). Hence, a declaration can have its type changed multiple times as the native and elaborated types are propagated down the parse tree.

A `typedef` can also be used to declare a type that is an array, as illustrated by the global type `char_buf` in Figure 10. In this case, the dimensions of the array given in the `typedef` declarator are propagated along with the type that the `typedef` aliases. The array dimensions are appended to the end of the declaration being flattened to preserve the semantics of the original source code. An example of this is shown in Figure 10.

Step 7: Unique Global Declarations. As described in Section 2.1, all preprocessor statements are removed from the file prior to transformation by TXL. However, this can often lead to source code that is no longer valid semantically. The preprocessor statements act as markers to GCC about where the original files, included via `#include` directives, start and stop. This allows GCC to track the file scopes that prevent re-declaration errors of function prototypes from occurring. For example, a file can include the GNU C standard library to have access to the function prototype for `malloc()`. If that same file also declares its own function prototype for `malloc()` (e.g., as an `extern`), then a re-declaration error can occur at compile time.

To account for this change in the file, Gemini scans over the global function and variable declarations in the file and removes duplicates. The end result is that only one function prototype for each function definition and only one `extern` declaration for each variable will remain at the end of the transformation.

4 Conclusion

The syntax of GNU C is very flexible. It gives the developer many ways to declare variables and types that are semantically equivalent, but the cost of this flexibility is ambiguity while parsing C source code.

Production GNU C code is difficult to automatically transform due to the presence of GNU C preprocessor statements. These statements give the developer a way to create macros and make other compile-time decisions. However, since these statements are not part of the GNU C syntax, they must be removed prior to transformation. Since the compiler is expecting these statements, removing them changes the semantics of the source code.

To perform automatic transformation on GNU C source code, many normalizing efforts must transpire. The normalization process ensures predictable formats of declarations and statements, and removes ambiguities that would otherwise make automated transformation very difficult.

References

[1] J. R. Cordy, T. R. Dean, A. J. Malton, and K. A. Schneider. Source Transformation in Software Engineering Using the TXL Transformation System. *Journal of Information and Software Technology*, 44(13):827–837, October 2002.

[2] J. R. Cordy, C. D. Halpern, and E. Promislow. TXL: A Rapid Prototyping System for Programming Language Dialects. In *Proceedings from IEEE International Conference on Computer Languages*, October 1988.

[3] C. Dahn and S. Mancoridis. Using Program Transformation to Secure C Programs Against Buffer Overflows. In *Proceedings of the Working Conference in Reverse Engineering (WCRE'03)*, 2003.

[4] Gemini, http://serg.cs.drexel.edu/gemini/.

[5] TXL homepage, http://www.txl.ca/.