Preprocessor Conditional Removal by Simple Partial Evaluation

Ira D. Baxter
Michael Mehlich
Semantic Designs, Inc.
www.semdesigns.com
{idbaxter,mmehlich}@semdesigns.com

Keywords
Partial evaluation, symbolic computation, preprocessor, program transformations

Abstract

Preprocessors with conditionals are often used as software configuration management tools, with preprocessor variables naming configuration aspects. The preprocessor conditionals found in large systems often have complex enabling Boolean equations and nesting. Over long periods, some configuration aspects lose all utility. Removal of all traces of a configuration variable and code dependent on that aspect becomes a significant task if performed manually.

An industrial-strength transformation system can remove such configuration aspects in a much shorter period of time, by symbolically simplifying away the configuration aspect. This paper sketches the problem, and the required transforms, and discusses a case study involving over a million lines of source code.

1 Removing Preprocessor Conditionals

Preprocessor directives are ubiquitous in a large software system, primarily existing to help configure aspects of the system for the variety of environments in which it runs to amortize its development cost. Typically these directives select language dialect, operating system or libraries (or versions thereof), application features, performance features, CPU type, word size, etc.

Different concerns are often (but not always) independent; the operating system chosen is usually independent of application features. We call each of these semi-separate concerns a configuration *aspect.* Such aspects are usually enabled or disabled in a binary fashion.

The success of a software system shows in its longevity. That longevity ensures that system configurations that were once valuable (e.g., VAXen) eventually become useless. Maintaining useless configurations in a large system increases its cost, as maintainers have to continually work around, understand, or even fix code for features they suspect, but are not sure, are no longer useful. Once a configuration aspect is declared dead, it would be ideal to remove everything from a source system that is related to that aspect.

Person-weeks of effort are required to do this manually for systems of scale (e.g., millions of source lines with thousands of files), and the work is unrewarding for the assigned individual.

Some kind of automation is needed. This should be done in a way that preserves as much of the source code structure (other preprocessor directives, indentation, spacing, number formats, comments, etc.) as possible, to prevent objections from programmers that "own" the code.

Ad hoc automated tools for this problem exist, usually based on string processing. These *almost* work, but are not good solutions because they do not really understand the full language syntax. Consequently, they can make errors at a significant rate on large systems, requiring the very manual intervention they were designed to avoid.

This work uses DMS, an industrial-strength transformation system. (DMS is intended to eventually support design maintenance [1,2,3], but is presently used for reengineering applications [4]). DMS accomplishes the desired effect by implementing partial evaluation of conditional preprocessor directives as a set of rewrite rules. Such transformation systems do understand the language syntax, and so can avoid the problems of ad hoc solutions. The rewrites implement what amounts to a Boolean expression simplifier and dead-branch elimination from `if-then-else` constructs. The value of DMS is its ability to parse difficult languages such as C and C++ in the presence of preprocessor directives, and the straightforwardness of the rewrites for accomplishing the desired effect.

Section 2 discusses preprocessors. Section 3 discusses the problems in preprocessor directive removal. Section 4 discusses industrial-scale transformation systems. Section 5 shows how source-to-source transformations are encoded and operate. Section 6 shows how we handle parsing preprocessor directives. Section 7 discusses practical issues when parsing preprocessor directives. Section 8 describes how partial evaluation can be used to simplify constructs. Section 9 shows the source-to-source transformation rules needed to remove the dead preprocessor directives. Section 10 shows the results, and Section 11 discusses future work.

2 Preprocessors

Many programming language environments provide "preprocessor" facilities, allowing considerable automated "editing" of a source file "before" it is seen by the language processor. Preprocessors with conditionals are often used as compile-time software configuration management tools, with preprocessor "variables" naming configuration aspects. These aspects can be Boolean (enabling or disabling a feature) or parametric (e.g. a integer specifying a `buffersize`), usually with a parameter value that by convention disables the feature (e.g., `buffersize == 0`).

```
#ifndef Unix
#define Unix 1
#endif
                    …
char filename[132];
#if SDOS
short int result[256];
int status;
#endif
                    …
#if (SDOS|Unix)&!DEMOS
  append_to(&filename,&extension);
#endif
#if SDOS
   syscall(OS_openfile,&filename,
           OS_readonly,&status,&result,);
   if (status!=OS_no_error)
      syscall(OS_exit_with_error,
              &status,OS_dummy,
              &status,&result);
#elsif Unix
   if (errno=fopen(&filename,readmode))
      { exit(errno); }
#else // DEMOS
                    …
#endif
```

Figure 1: Source code with preprocessor directives

Figure 1 shows typical configured source code. The preprocessor variables `Unix`, `DEMOS` and `SDOS` are aspects controlling which operating system calls are made. By configuration, we mean all source code defining or dependent on an aspect; for `SDOS`, this is the conditional declaration of a `result` buffer, the filename extension-appending code, and the "then" branch of the file opening logic. Note that configurations often overlap because of shared logic, such as appending the extension.

The configuration space managed can be enormous. N Boolean configuration variables can provide for up to 2^N possibly different instantiated configuration; the Linux operating system has roughly 1000 configuration variables [7]. The preprocessor conditionals found in large systems often have complex enabling Boolean equations over these aspects, which encode configuration dependencies, and complex nesting to allow code sharing between configurations.

The term "preprocessor" comes from the fact that these facilities have traditionally been grafted onto languages as an afterthought, and consequently implemented as a "preprocessing" pass to perform the "editing", and occurring before the compilation step (most Unix systems and many "C" compilers). For efficiency reasons, preprocessing is often now integrated into the compile step, but the name remains.

The preprocessor provides, via special "preprocessor directives" easily found in the program text, several types of facilities that change the apparent source file:

- importing fixed blocks of text (C `#include` files)
- inserting parameterized blocks of text (C macro invocations, COBOL copybooks)
- selecting or deselecting existing blocks of text (including other preprocessor directives) based on some configuration variable or expression (e.g., C `#if`, `#elsif`, `#else`, `#endif` directives)
- defining blocks of text for later inclusion (C macro `#define`)
- marking certain identifiers used as configuration variables as defined (C `#define` with empty macro) or undefined
- testing the definedness status of such configuration variables (C `#ifdef`, `#ifndef`, `defined`, `undefined`)
- combining two or more language lexemes to create a new lexeme. This is almost always used to construct a "gensym" identifier (C macro body operation ##).

C, C++ and COBOL85 all provide a preprocessor facility defined by the language standard, based on the language lexemes. Users of languages without native preprocessing facilities often fall back on the C preprocessor (if the lexeme structures are close enough, e.g. FORTRAN), custom preprocessors (e.g., Generic PreProcessor [6]) or on ad hoc solutions modeled after those for C, often based on a string processing tool like PERL.

Ada83 (and Ada95) were specifically designed without a preprocessor on the grounds that such editing can be done by simply choosing, outside the scope of the language, which source components to use. However, this artifice forces large grain source components that replicate considerable content, creating the very maintenance

problem that preprocessor conditionals were meant to eliminate. The result is that the argument over whether Ada should have a preprocessor continues to this day, with the many defectors forced to use an ad hoc solution.

Another solution integrates the "preprocessor" directives and conditionals directly into the syntax of the programming language [9]. Unlike string or lexeme based preprocessors, syntax-based preprocessors are only allowed in a small number of places, such as declarations, statements and expressions; nonetheless, this solution provides all the essential configuration power needed. This method was inspired by the difficulties of parsing languages containing preprocessor directives.

In any case, preprocessor directives have repeatedly proven their utility for "configuration management in the large" to enable the economical construction of large scale software systems running on many platforms. Compile-time conditional configuration (misnamed "preprocessing" because of C's implementation) is a permanent feature of the large systems landscape.

3 Removing Dead Configurations

Most large software systems have a lifetime of ten years or more, and there are banking, insurance and military applications with life spans approaching 50 years. Any long-lived artifact lives in an evolving world, and evolves with it. New configuration demands arrive, and old ones fade away.

Our concern is with older configuration aspects that lose all utility, because the aspect addresses circumstances that no longer occur. Paper tape support is no longer needed; Digital Equipment no longer manufactures VAX processors; and 8 bit operating systems vanish (SDOS in Figure 1; Microsoft has announced the death of MSDOS).

Maintaining dead configurations in a large system increases its cost, as maintainers have to continually work around, understand, or even fix code for features they suspect, but are not sure, are no longer useful. Such useless configurations may even be declared as dead by the system architects, and known by the grizzled programmers as useless. Once a configuration aspect is declared dead, it would be ideal to remove everything related to it from a source system that is related to that aspect. Otherwise, programmers, spending typically 50% of their time simply looking at source code, will spend precious time just to ignore the configuration, sometimes will waste precious time trying not to disturb the configuration, and finally, occasionally will actually use time fixing or enhancing the configuration in the mistaken belief that it is still useful.

Attempts to remove such configuration variables by manual methods is often frustrated by scale, time pressures, and the sheer dullness of the task. For a million SLOC C program with 1000 source files, assuming 15 minutes per file to check out, edit, compile, debug the inevitable occasional mistaken change, and regression-test, it takes 11 person-weeks of effort to make a single pass. Careful attention is required to make sure that shared configuration code is properly adjusted; in Figure 1, it is all to easy to erroneously delete the entire conditional block containing the file-appending logic when removing the `SDOS` configuration, and testing under `DEMOS`, since no compile-time error will occur. This problem is compounded by the fact that several configurations may die every year. Most software engineers are not rewarded for doing such work, as the management perception is that there is always more pressing work to do, and the programmer perception is that the work is extremely repetitive.

The inevitable conclusion is that an automated tool is needed to remove dead configurations. This should be done in a way that preserves as much of the source code structure (other preprocessor directives, indentation, spacing, number formats, comments, etc.) as possible, to prevent objections from programmers that "own" the code.

Removing dead configurations is essentially a problem of removing dead code, a well-understood compiler problem. However, one cannot use conventional compilers for this problem, as they perform dead code elimination on generated object code, in which the source, comments, and all preprocessor directives have been lost.

Ad hoc solutions to this problem exist, apparently based on string processing, such as UNIFDEF, RMIFDEF, CPPP (based on PERL), SCPP ("Selective C preprocessor"), but are not good solutions because they do not really understand the full language lexical and grammatical syntax; their very nature makes them difficult to complete. Consequently, these tools can make significant errors, requiring the very manual intervention they were designed to avoid. For example, UNIFDEF does not handle `defined` or `#elif` preprocessor directives Some of these tools leave complex conditionals in the code because they do not obviously simplify to TRUE or FALSE, retaining the very configuration variables they should be removing. Arguably, these defects could be repaired with enough effort. What cannot be easily repaired are more complex problems, such as determining that both arms of a conditional assign the same value to a preprocessor variable, i.e., invariant results. We do not handle this either, but our approach can be extended to handle it, and to integrate with other reengineering activities.

An industrial-strength transformation system can more reliably remove such configuration aspects, by symbolically simplifying away the dead configuration aspect.

4 Industrial-Strength Transformation Systems

By industrial strength transformation system, we mean ones that:

- Accept language definitions for real languages
- Can parse and prettyprint those languages on scale
- Accept source-to-source rewrite rules in those languages
- Can apply those rules to a source base reliably
- Have been applied in practice

We know of only a few such systems:

- REFINE (available commercially from Reasoning Systems: www.reasoning.com)
- TXL (available via www.txl.ca, formerly from Legasys)
- DMS (available commercially from Semantic Designs: www.semdesigns.com)

These tend to be commercial systems because of the effort it takes to implement them (the present DMS has well over 50 person-years invested directly in the engineering). There are a host of other transformation systems, many listed at www.program-transformation.org. (The author apologizes to any that meet the criteria but are not listed here, and would appreciate knowing about them.)

Many compiler toolkits (e.g. YACC) offer LL(1) or LALR(1) parsers, which work by definition only for very limited classes of languages. Most compilers and tools tend to have parsers with ad-hoc modifications to step around parser limitations. This means the compiler infrastructure is not good for a wide range of languages. What one needs is a full context-free parsing mechanism, which both TXL and DMS have (DMS uses GLR (aka Tomita) parsers [11, 10]).

What distinguishes industrial-strength transformation systems from compiler toolkits is: the configurability and robustness of their parsing technology, the integration of that parsing technology with the pattern languages used for source-to-source rewriting, and the ability to regenerate legal source programs in all necessary details from internal representations (ASTs). This capability is used for large scale reengineering (e.g., code porting), software quality analysis and enhancement (e.g., clone detection and removal [4]), etc.

To use these transformation systems, the language syntax of interest has to be defined. Because these tools are highly configurable, this is far less of a task than building a compiler front end. Further, these systems are often available with predefined language modules for mainstream languages, such as C, C++, Ada, Fortran, etc.

Without such foundations, it is uneconomical to build program transformation tools for legacy languages or large-scale software systems.

5 Source to Source Transformations

Source to source program transformations ("rewrite rules") are used to modify programs directly in terms of the programming language syntax. (Other program transformations may be implemented by procedural code, or sets of transformations). These rewrites are usually stated in terms derived from an abstract or concrete grammar, and these terms in turn correspond to underlying abstract syntax trees.

A typical rewrite rule abstractly has the following form:

LHS → *RHS* **if** *condition*

where both *LHS* ("left hand side") and *RHS* ("right hand side") represent source language patterns with variables to represent arbitrarily long well formed language sub strings. The **if** *condition* is an optional phrase referring to the variables in the LHS pattern. These rules are interpreted as, "when a program part matches the *LHS*, replace it by the *RHS*, if *condition* is true". The condition may be implemented as some additional matching constraints, or a call on some decision procedure.

Real transformations systems add more syntax to this simple scheme to allow specification of more details about the patterns. For DMS, an example rewrite on C code to convert an assignment statement into an auto-increment is shown in Figure 2. This rule is written in DMS's *Rule Specification Language*. (For these examples, we take advantage of the availability of a C language module for DMS. We also take slight liberties with the transforms to simplify their presentation).

```
default domain C.

rule auto_inc(v:lvalue):
    statement->statement =
  "\v = \v+1;" rewrites to "\v++;"
       if no_side_effects(v).
```

Figure 2: A DMS rewrite rule

This defines a rewrite `rule` with name `auto_inc`, having a syntax variable `v` of syntactic class `lvalue`. The rule is a map from a C `statement` to a C `statement` (maps from one syntax class to another, and maps from one language to another are also possible with DMS). The text inside the quotation marks is legal C source, modulo the possibility of escaped rule language tokens (marked with \), such as syntax variables (e.g.,

`\v)`, which stand for arbitrary legal C source satisfying the syntactic category `lvalue`. The left hand quoted string is the rule *LHS*, and the right hand quoted string is the *RHS*. The occurrence of the same syntax variable multiple times in the *LHS* requires that the same exact sequence occur in both places; this is how the rule is constrained to match identical source and target. The occurrence of the syntax variable in the *RHS* requires that the changed program include what was matched for that variable on the *LHS*. Finally, the **if** condition in this rule is a language-dependent decision procedure that makes sure that the program fragment matched by \v contains no side effects, which would make this transformation incorrect.

Before rule use, a typical rewriting engine first parses the rule according to its rule language, and then parses the quoted pattern in the language to be transformed (here specified by the `default domain` phrase as the "C" language), to construct pattern trees. At transformation time, the rewrite engine matches the *LHS* pattern tree against portions of the program, and replaces matched trees by the corresponding *RHS* tree if the *condition* is satisfied. The Figure 2 rule has the effect shown in Figure 3.

before: `(*Z)[a>>2]=(*Z)[a>>2]+1;`

after: `(*Z)[a>>2]++;`

Figure 3

Typically a transformation system will have a large number of rules, and a large number of possible places in a program to apply them. It is beyond the scope of this paper to describe how the transformation system chooses which rules and where to apply them. The simple notion that all rules possessed are applied leaf-upwards to the entire parse tree for a file is adequate for this paper, and supported directly as one mode of operation of DMS.

DMS captures comments and number literal formats (radix, leading zeros, etc) while lexing and attaches them to the trees while parsing, and is later able to reproduce the source program in its original- or transformed-form from the trees with all the essential formatting information. Further discussion of the DMS prettyprint capability is outside the scope of this paper. Comments attached to undisturbed trees, and to the roots of rewritten trees are preserved, and so comments are not generally lost or changed by rewriting. (One can always build special rules that change this behavior).

6 A Simplified C grammar

In this section we briefly sketch a simplified version of the "C" grammar we use, because the rewrite rules are defined in terms of grammar tokens. Arguably, we could have defined only a "preprocessor" language, treating the regions between preprocessor directives as uninterpreted text, but it is more economical to amortize our reengineering investments by operating on a full C grammar. Such a full grammar enables other reengineering activities to be implemented, such as source reformatting, clone detection, refactoring, etc. Figure 12 shows a subgrammar for preprocessor expressions.

We wish to parse "C" source *before* preprocessor expansion occurs. However, C preprocessor directives can occur between any pair of tokens. One can code a grammar for that, but it is extremely unwieldy and therefore impractical.

Instead, we allow preprocessor directives in the grammar only at places where they commonly occur in real programs. Figures 10 and 11 show subgrammars, which allow directives only at file scope and in lists of statements, respectively. Allowing directives in other places in the grammar is straightforward: replicate the preprocessor conditional subgrammar and instantiate for that place type. In the actual grammar we allowed them in locals lists, and inside expressions and several other places.

7 Parsing Unpreproccessed C

There are a number of other complications that occur trying to parse C this way. We have a special preprocessor that delays expanding directives if it can and collects information from include files without expanding them in place. "Different" values of a `#defined` variable may reach the end of a preprocessor conditional. Macro calls often ambiguously look like function calls and have possibly to be resolved later. Discussion of how all this is handled is beyond the scope of this paper.

```
if (. . .)
      { . . .
#if (. . .)
       } else {
#endif
     . . .
      };
```

Figure 4: Badly nested preprocessor directives and language conditionals

With sundry additional help in place, DMS parses typically 85% of unpreprocessed C source files in large, real source systems directly, producing ASTs containing preprocessor conditionals. The unparsable balance tends to use preprocessor directives in unstructured ways. Figure 4

shows one example, in which the nesting of the preprocessor conditional is inconsistent with the nesting of the `if` statement. The reaction of most staff to this kind of trick is first, horror, and then second, to insist on removing the trick from the source. Doing such repairs on 50K SLOC of C code can take an afternoon. So with some motivated code cleanup help, we are able to read large systems. Of course, tricky idioms common to a system can simply be added to the grammar, and then transformed away.

8 Partial Evaluation

A method for simplifying programs when program parameters are known is called partial evaluation [8]. The basic idea is that computations having one or more constant operands often have algebraic rules that show equivalence to often much simpler computations (e.g., `pow(x,n), the` C idiom for x^n, when `n` is `2`, can be replaced by `x*x`). Often, there is a cascade of simplifications when a complex operation decays into a constant, feeding further simplifications (e.g, `x/pow(x,n)` when `n` is `1` reduces to just `1` for nonzero `x`).

An industrial strength transformation system can encode the basic partial evaluation rules directly, in the surface syntax forms of the language being simplified. Figure 5 shows the rules from the previous paragraph.

```
rule simplify_x_squared(e:exp):
   product-> product =
    "pow(\e,2.)" rewrites to "\e*\e"
   if no_side_effects(e).

rule simplify_divide_by_1(e:exp):
   product->product =
   " \e / 1 " rewrites to "\e".
```

Figure 5: Some partial evaluation rules in C

Figure 6 shows partial evaluation rules for Boolean expressions in preprocessor conditionals. These handle partial evaluation of Boolean expressions in preprocessor expressions. We don't need to write rules for all combinations of commutative operators (e.g., `"0 || \e"`) because we have declared the grammar rules that correspond to those operators as `associative-commutative`, and so the matcher knows to try the other orders.

Sophisticated partial evaluation for procedural languages often requires data flow analysis to determine how optimized results in one part of the program may enable optimizations in another part, such as when a constant is assigned to a variable used and updated in many other places in the code. For preprocessor simplification, we do not need much of this, as most preprocessor variables are assigned once, and preprocessor conditional expressions are completely functional. In fact, the DMS preprocessor does a kind of dataflow analysis of preprocessor values across preprocessor conditionals, but that discussion is outside the scope of this paper. More importantly, no preprocessor variable assignment inside a conditional can affect what the conditional or its containing parents will do.

```
rule or_true(e:ppexp): ppexp->ppexp:
  "\e || 1" rewrites to "1".

rule or_false(e:ppexp): ppexp->ppexp:
  "\e || 0" rewrites to "\e".

rule and_true(e:ppand): ppand->ppand:
  "\e && 1" rewrites to "\e".

rule and_false(e:ppand): ppand->ppand:
  "\e && 0" rewrites to "0".

rule not_true(): ppterm->ppterm:
  "! 1" rewrites to "0".

rule not_false(): ppterm->ppterm:
  "! 0" rewrites to "1".

rule parenthesesK(n:NATURAL):
   ppexp->ppexp:
  "( \n )" rewrites to "\n".

ruleset boolean_partial_evaluate =
  { or_true, or_false,
    and_true, and_false,
    not_true, not_false,
    parenthesesK }.
```

Figure 6: Preprocessor expression Boolean equation partial evaluation

We implement preprocessor conditional removal as partial evaluation using rewrite rules, with known constant values for selected configuration aspects. We count

```
rule delete_SDOS(): ppexp->ppexp:
  "SDOS" rewrites to "0".

rule fast_newline(): ppexp->ppexp:
  "CRLFDelay" rewrites to "0".

rule one_display(): ppexp->ppexp:
  "NumberOfScreens" rewrites to "1".

ruleset clean_up =
  { delete_SDOS, no_newline_nulls,
    one_display_only }.
```

Figure 7 Rewrites encoding future permanent state of dead configuration aspects

heavily on chains of simplification to carry out the actual removal process.

Application of partial evaluation on abstract C preprocessor directives was considered by [5], but he did not build a running system.

9 Rewrites for Preprocessor Simplification

We need three sets of rules to remove useless preprocessor directives:

1) Custom rewrites specifying which configuration parameters are permanently set
2) Boolean algebra partial evaluation for the constants true and false (Figure 6).
3) Preprocesser conditional simplification for true and false conditions.

The first set is shown in Figure 7. Their purpose is to specify the permanent future status of the configuration parameter(s) that we wish to remove, and are determined by the specific application we wish to revise. The syntax class `ppexp` refers to "preprocessor expressions", which are distinct from the syntax class for normal C expressions, `expr`. We package the individual rules into a rule set, to make it easier to apply them all as a group. In general, this rule set is likely to be a very small set, because we discover the need to remove configurations rarely, encode the set of interest, and then run the removal process.

The preprocessor Boolean condition rewrites in Figure 6 is the second set. A subset of the third set of rewrites are shown in Figures 13 and 14. These are required to partially-evaluate preprocessor conditionals when the condition status is known to be false (`0`) or true (`1`). The number of rules is explained by the combinatorial properties of `#if`, `#elsif` and `#else` in the grammar. In fact, we only show the rules for preprocessor directives allowed at file scope.

```
#ifndef Unix
#define Unix 1
#endif
                    …
char filename[132];
                    …
#if (Unix)&!DEMOS
  append_to(&filename,&extension);
#endif
#if Unix
   if (errno=fopen(&filename,readmode))
      { exit(errno); }
#else // DEMOS
                    …
#endif
```

Figure 9: Preprocessor simplification result

Nearly identical rules are required for preprocessor conditionals allowed at other points in the grammar, such as at statement-list level. The Boolean rewrites and preprocessor conditional simplification rules are used for every preprocessor removal.run.

10 Application and Results

Using the tools to remove conditionals is straightforward. The dead configuration aspects are coded as rules, added to the preprocessor partial evaluation rules, and supplied as the rules base for DMS to use (Figure 8). The source file(s) of interest are parsed using the C grammar definition, the rules are also parsed using that definition, and then DMS applies the rules to carry out the partial evaluation with respect to the dead configuration aspects. The result of applying Figure 6 to Figure 1 is shown in Figure 9.

For ANSI C, the actual set of rewrites is rather larger than is shown here because of the number of places

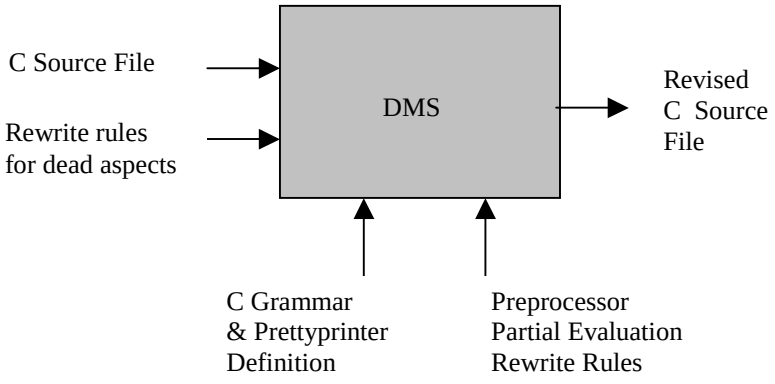

Figure 8: Preprocessor Removal Process

preprocessor conditionals are allowed in the grammar. Nonetheless, it took one-person day to encode several hundred similar rules, and one day to test the complete set (of course, we had already built the C grammar for other reengineering activities, and paid the cost of learning how to use the tools). DMS was then applied to a commercially successful software system of 1.5 million SLOC in 1800 files, and carried out fully automated preprocessor removal for about 80% without intervention. The remaining files required hand-remediation of unstructured preprocessor directives before automated removal was successful. The customer for whom this work was done estimated that 10 weeks was removed from a very busy schedule.

11 Future Work

For languages having real "pre" processors, there are still some problems when the preprocessor conditionals fail to nicely nest with the language structures. Our practical experience is that these problems can be fixed manually at modest cost, and those fixes will make the code not only more understandable to the software engineers, but make it that much easier to apply automated tools for the next interesting task. However, being able to handle these cases would make the tools materially easier to apply on large systems. We plan to parse such fragments as a stream of trees, with an assembly constraint corresponding to the nonterminal around the largest unparseable chunk. It is our understanding that this may not work with sophisticated preprocessor usage possible with the PL/I preprocessor.

One can avoid this difficulty in new language designs, by integrating the "preprocessor" directives at suitable places in the language syntax. We have already done that with our internal parallel-processing language, PARLANSE, with benefits accruing in simplified parsing and analysis.

Our present tool also does not handle cascaded preprocessor assignments. As a consequence of removing a conditional from around a line such as `#define FOO 1`, FOO becomes clearly a known constant, and further simplifications can occur. When a conditional is removed, it is straightforward to rescan the suddenly unconditional trees to detect such assignments. One could then use a procedural transform to implement the additional configuration aspect rewrite (i.e., substituting the value), thereby enabling possibly further simplifications.

Conclusion

We considered preprocessors as compile-time text editors, and the permanent role they play in economically configuring large systems for many platforms. We have discussed the problem of removing defunct configuration aspects by removing their implementing preprocessor conditionals from real source codes by using partial evaluation methods.

To carry this out in practice, one needs industrial strength program transformation systems, such as DMS, that allow one to specify the target language, the basic partial evaluation rules, and rules that specify which configuration aspects are to be removed.

These tools have been automatically applied to large suites of C code, producing correct results in a few hours, saving large amounts of manual modification, and saving future maintenance costs. Such tools can easily be reconfigured for other programming languages having preprocessor directives.

Such industrial strength transformation systems promise to make massive software modifications much more practical, and therefore much more common.

References

[1] I. Baxter, "Design Maintenance Systems", *Comm. of the ACM 35(4)*, ACM, April 1992.

[2] I. Baxter and M. Mehlich, "Reverse Engineering is Reverse Forward Engineering". *4th Working Conference on Reverse Engineering*, IEEE, 1997.

[3] I. Baxter and C. Pidgeon, "Software Change Through Design Maintenance". *Proc. International Conference on Software Maintenance*, IEEE, 1997.

[4] I. Baxter, et al, "Clone Detection Using Abstract Syntax Trees", Proc. *International Conference on Software Maintenance*, IEEE, 1998.

[5] J. Favre, "A Rigorous Approach to Support the Maintenance of Large Portable Software", *Proc. Conference on Software Maintenance and Reengineering*, IEEE, 1997.

[6] http://math.polytechnique.fr/cmat/auroux/prog/gpp.html

[7] Y. Hu, E. Merlo, M. Dagenais and B. Lague, "C/C++ Conditional Compilation Analysis Using Symbolic Execution", *Proc. International Conference on Software Maintenance*, IEEE, 2000.

[8] N. Jones, et al, *Partial Evaluation and Automatic Program Generation*, Prentice-Hall 1993

[9] *PARLANSE Reference Manual*, Semantic Designs, 1998.

[10] M. Tomita, *Efficient Parsing for Natural Languages*, Kluwer Academic Publishers, 1988.

[11], M van den Brand, et al, "Current Parsing Techniques in Software Renovation Considered Harmful", *Proc. Sixth International Workshop on Program Comprehension*, IEEE, 1998.

Simplified C grammar

```
program = decl_list ;

decl_list = ;

decl_list = decl_list decl ;

decl = var_declaration ;

decl =
   type IDENTIFIER ‘(‘ params ‘)’
        ‘{‘ body ‘}’;

decl =
   ‘#’ ‘if’ ppexp ppNL
       decl_list
   ppthen_decl ;

ppthen_decl = ‘#’ ‘endif’ ppNL ;

ppthen_decl =

   ‘#’ ‘else’ ppNL ;
        decl_list
   ‘#’ ‘endif’ ppNL ;

ppthen_decl =
   ‘#’ ‘elsif’ ppexp ppNL
       decl_list
   ppthen_decl ;
```

Figure 10: Simplified C:
File scope subgrammar

```
body = locals stmt_list ;

stmt_list = ;

stmt_list = stmt_list stmt ;

stmt = exp_stmt ;

stmt = ‘if’ ‘(‘ exp ‘)’ stmt ;

stmt = ‘{‘ body ‘}’ ;

stmt =
   ‘#’ ‘if’ ppexp ppNL
       stmt_list
    ppthen_stmt ;

ppthen_stmt = ‘#’ ‘endif’ ppNL ;

ppthen_stmt =

   ‘#’ ‘else’ ppNL ;
        stmt_list
   ‘#’ ‘endif’ ppNL ;

ppthen_stmt =
   ‘#’ ‘elsif’ ppexp ppNL
        stmt_list
    ppthen_stmt ;
```

Figure 11: Simplified C:
Statement list subgrammar

```
ppexp = ppand ;


[associative-commutative]
    ppexp = ppexp ‘|’ ppand ;

ppand = ppsum ;


[associative-commutative]
   ppand = ppand ‘&’ ppreln ;


ppreln = ppterm ;

ppreln = ppterm ‘==’ ppterm ;

ppterm = NATURAL ;

ppterm = ‘(‘ ppexp ‘)’ ;

ppterm = ‘!’ ppterm ;
```

Figure 12: Simplified C:
Preprocessor expression subgrammar

Note: `ppNL` is the name of the C grammar token for a preprocessor new line.

Rules to simplify preprocessor conditionals

```
rule decl_if_true_endif
    (d:decl_list): decl->decl =
  "#if 1 \&n \d #endif \&n"
     rewrites to "\d".

rule decl_if_false_endif
    (d:decl_list): decl->decl =
  "#if 0 \&n \d #endif \&n"
     rewrites to ";".

rule decl_if_true_else
    (d1:decl_list,d2:decl_list):
    decl->decl =
  "#if 1 \&n \d1
   #else \&n \d2 #endif \&n"
     rewrites to "\d1".

rule decl_if_false_else
    (d1:decl_list,d2:decl_list):
    decl->decl =
  "#if 0 \&n \d1
   #else \&n \d2 #endif \&n"
     rewrites to "\d2".

rule decl_if_true_elseif
    (d1:decl_list,e:ppexp,
     d2:decl_list,rest:ppthen_decl):
    decl->decl =
  "#if 1 \&n \d1
   #elsif (\e) \&n \d2 \rest"
     rewrites to "\d1".

rule decl_if_false_elsif
    (d1:decl_list,e:ppexp,
     d2:decl_list,rest:ppthen_decl):
    decl->decl =
  "#if 0 \&n \d1
   #elsif \e \&n \d2 \rest"
     rewrites to
  "#if \e \&n \d2 \rest".
```

Figure 13: `#if` partial evaluation rules

Note: inside DMS patterns, actual newline characters are treated as whitepsace. `\&n` is an escape encoding an actual newline character

```
rule decl_elsif_true_endif
    (d:decl_list):
    ppthen_decl->ppthen_decl =
  "#elsif 1 \&n \d #endif \&n"
     rewrites to
  "#else \&n \d #endif \&n".

rule decl_elsif_false_endif
    (d:decl_list):
       ppthen_decl->ppthen_decl =
  "#elsif 0 \&n \d #endif \&n"
     rewrites to "#endif \&n".

rule decl_elsif_true_else
    (d1:decl_list,d2:decl_list):
    ppthen_decl->ppthen_decl =
  "#elsif 1 \&n \d1
   #else \&n \d2 #endif \&n"
     rewrites to
  "#else \&n \d1 #endif \&n".

rule decl_elsif_false_else
    (d1:decl_list,d2:decl_list):
    decl->decl =
  "#elsif 0 \&n \d1
   #else \&n \d2 #endif \&n"
     rewrites to
  "#else \&n \d2 #endif \&n".

rule decl_elsif_true_elsif
    (d1:decl_list,e:ppexp,
     d2:decl_list,rest:ppthen_decl):
     decl->decl =
  "#elsif 1 \&n \d1
   #elsif \e \&n \d2 \rest"
     rewrites to
  "#elsif 1 \&n \d1 \rest".

rule decl_elsif_false_elsif
    (d1:decl_list,e:ppexp,
     d2:decl_list,rest:ppthen_decl):
     ppthen_decl->ppthen_decl =
  "#elsif 0 \&n \d1
   #elsif \e \&n \d2 \rest"
     rewrites to
  "#elsif \e \&n \d2 \rest".
```

Figure 14: `#else` partial evaluation rules