Rewrite Systems for Symbolic Evaluation of C-like Preprocessing

Mario Latendresse

Northrop Grumman IT
Technology Advancement Group/FNMOC/U.S. Navy
7 Grace Hopper, Monterey, CA, USA 93943
E-mail: mario.latendresse.ca@metnet.navy.mil

Abstract

Automatic analysis of programs with preprocessing directives and conditional compilation is challenging. The difficulties range from parsing to program understanding. Symbolic evaluation offers a fundamental and general approach to solve these difficulties. It finds, for every line of code, the Boolean expression under which it is compiled or reached. It can also find all the possible values of preprocessing variables (macros) for each line of code. Conditional values have been shown an effective representation to do fast practical symbolic evaluation of preprocessing; but their interaction with macro expansion and evaluation has not been formally investigated. We present convergent rewrite systems over conditional values that can interact with macro expansion and evaluation and transform them into Boolean expressions. Once transformed, well known simplification techniques for Boolean expressions can be applied. This entails a more complete solution to the efficient symbolic evaluation of C-preprocessing using conditional values.

1 Introduction

Textual preprocessors similar to `cpp` might be considered obsolete and ill-designed tools, but they are still widely used in practice from small to large software projects. C-like preprocessing, as described by ANSI C, and implemented by `cpp`, is a *de facto* approach for preprocessing not only for C but also for programming languages as diverse as Fortran and Haskell. Moreover the design of some textual preprocessors are similar to `cpp` [7, 11].

Many researchers [20, 8, 19, 16, 5, 6, 9] have described some of the difficulties of code analysis, maintenance and refactoring in the presence of such preprocessing. Indeed, conditional compilation, free preprocessing variables and macro expansion bring difficulties at many levels, from parsing to program understanding.

Several refactoring and visualization tools [20, 1, 15, 17, 12, 16] are based on ad hoc control-flow analyses of preprocessing—they would benefit from a precise and complete (non-abstract) control-flow analysis of conditional compilation in the presence of macro expansion.

As far as we know, all solutions to handle program analysis in the presence of preprocessing are based on heuristics; and they are often based on partial parsing. These solutions may be good enough for certain specific problems, but they still leave open a general approach capable of handling precisely the semantics of C-like preprocessing.

A precise, non-abstract, symbolic evaluation of C-like preprocessing is a promising approach since such concrete preprocessing is not Turing complete[1].

In [14] a symbolic evaluation technique was presented to provide a fundamental solution to automatic analysis of preprocessed code. It does not require the code to be parseable by a context-free grammar. Its direct goal is to find, for every line of code, the condition under which it is compiled or reached. It also provides, for every line of code, the possible values—under guarded Boolean expressions—of preprocessing variables[2].

Given such information, further analysis or transformations can be done. For example, all the statically dead code could be removed[3], all possible macro values at every line of code can be found, refactoring operations such as renaming of variables can be made precise, etc.

This symbolic evaluation uses conditional values (c-values)[4]. They were shown, in [14], to be effective to avoid

[1] This can easily be proven, since ANSI C preprocessing only possible form of iteration is the `#include` mechanism, which has an nested depth constraint.

[2] In practice, such information is not kept for every line of code but for every segment of code which is a sequence of lines without preprocessing conditionals.

[3] This should not always be done, since some parts of the code might intentionally be cut out temporarily.

[4] We might call them conditional expressions as they may be based on preprocessing variables and operators but since they are only bound to preprocessing variables we prefer the term ‘value’.

path analysis and its exponential time complexity of conventional symbolic evaluation.

Informally, a c-value $c\,?\,e_1 \diamond e_2$ may be bound to a preprocessing variable: if the condition c is true, its value is e_1 otherwise it is e_2.

The main objective of this paper is to formalize the manipulation of c-values in the pressence of macro-expansion, parsing and evaluation of if-directives; in particular, to present convergent rewrite systems for these three phases of preprocessing. In essence, these rewrite systems transform the c-values to Boolean expressions annotating the lines of code and the values of preprocessing variables.

We emphasize that, as far as we know, all works on symbolic preprocessing do not take in consideration these three phases such that a precise solution is obtained. In particular, the correct interleaving of macro-expansion, parsing and evaluation during preprocessing is essential: therefore, three rewrite systems are necessary.

In this paper we use the vocabulary and notations of [4] for rewrite systems.

Section 2 presents the formal syntax of terms, including c-values, and the symbolic evaluation algorithm. Section 3 discusses the identification of free preprocessing variables used in Boolean terms. Section 4 presents the concrete and symbolic expansion, parsing and evaluation of conditionals in the presence of c-values and the main syntax of our terms. The rewrite systems are presented in Section 5. Section 6 presents some rewrite examples. Comparisons to related works is presented in Section 7.

2 Symbolic algorithm and c-values

We reproduced in Figure 1 the symbolic evaluation algorithm presented in [14]; in particular the `Merge` procedure which generates the c-values.

It is based on the control flow graph (CFG) built from the source code and based on the `#if` directives. We assume that all include directives use only constant argument. In this manner the CFG can be built prior to preprocessing. Such an assumption is essential, otherwise some included files would be unknown from the source code. Consequently, the CFG does not contain any include directive. Note that, due to recursive file inclusion, the CFG may contain loops.

The objective of this algorithm is to find for every line of code the condition under which it is compiled or reached. It also finds all the possible values of every preprocessing variable. Complete examples can be found in [14].

In this paper, *concrete preprocessing* refers to the preprocessing done by the preprocessor (i.e., `cpp`); *symbolic preprocessing* refers to the preprocessing done by our symbolic evaluation algorithm.

Essentially, the symbolic algorithm traverses the CFG, keeps track of the definitions done by `#define` (line 12) on stack S of tables, evaluates the if-directive conditionals (line 13), merges new definitions into c-values and annotates blocks of code (i.e., code segments) with Boolean terms (line 8).

On line 13, the if-directive conditional is expanded which may then contain c-values; but it is "simplified" (rewritten) to remove them. This elimination is proven by the rewrite systems to be defined. All c-values are generated by the `Merge` procedure on line 30. Since v_t and v_f may be c-values themselves, c-values may be nested forming a tree.

Informally, the c-value $c\,?\,e_1 \diamond e_2$ bound to a preprocessing variable x means: if the condition c is true, the value of x is e_1 otherwise it is e_2.

We use two sets of terms for c-values. The first one, $\mathcal{T}_C$, handles the unparsed sequences of tokens and can interact with parsing and evaluation. The second one, $\mathcal{T}_{C\uparrow}$, applies to parsed and evaluated sequences of tokens and is used for the last phase of c-value rewriting.

The first c-value terms syntax is given in Figure 2. Such a c-value is either a preprocessing (concrete) value, the initial value of a free variable, or a guarded set of c-values structured as a tree. The formal syntax of concrete value is represented as terms and given in Figure 3. We defer discussion of free preprocessing variables, and their initial value, in Section 3. A c-value that is neither concrete nor the initial value of a free variable is represented as $c\,?\,e_1 \diamond e_2$, where c is a Boolean expression (a $\mathcal{T}_B$ term) and e_1, e_2 are c-values. The syntax of terms $\mathcal{T}_B$ are given in Figure 4.

Concrete values correspond to values that are generated during a concrete preprocessing. It is either a sequence of valid tokens, $\top$ (defined) or $\bot$ (undefined), generated respectively by the directive `#define` without a list of tokens and the directive `#undef`.

Note that, an unparsed c-value $\mathcal{T}_C$ can be interpreted as a tree whose leaves are either $\bot$, $\top$, a free preprocessing variable, or a sequence of valid tokens. These sequences of tokens are unparsed since parsing occurs during conditional evaluation after macro expansion. Such a representation allows the combination of c-values, whose leaves may be syntactically invalid, to form syntactically valid C-like conditional expressions.

For unparsed c-values, we will enclose sequences of tokens between single quotes as in $\texttt{def}(x_I)\,?\,\texttt{`2 + 4'} \diamond \texttt{`0'}$.

After the symbolic evaluation, every line of code, including directives, is annotated by a Boolean expression, hereafter called a *final Boolean term* as defined by the $\mathcal{T}_B$ terms. Note that their definition is not based on c-values—they do not contain any c-value.

The terms $\mathcal{T}_B$ include errors, represented by the domain Err, since different errors may occur during parsing and

1. **Main**
2. Push empty table [] onto S
3. **Call** $V(A, \texttt{true})$
4. The CFG A contains all conditions
5. The table in S has the final variable bindings
6. **End**

7. **Procedure** $V(n, c_c)$ {
8. add c_c to condition list of node n;
9. test node n for possible infinite iteration;
10. **Case** node n
11. block of code: nothing to do;
12. define: add definition to top table of S;
13. if: Let c be its expanded/simplified condition
14. **if** $c_c \wedge c$ is satisfiable **then** {
15. Push empty table [] onto S;
16. **Call** $V(n.then, c_c \wedge c)$;
17. Pop top table from S and assign it to T;
18. } **else** T is empty;
19. **if** $c_c \wedge \neg c$ is satisfiable and $n.else$ exists **then** {
20. Push empty table [] onto S;
21. **Call** $V(n.else, c_c \wedge \neg c)$;
22. Pop top table from S and assign it to E;
23. } **else** E is empty;
24. Merge(T, E, S, c);
25. **End Case**
26. **if** $n.next$ exists **then Call** $V(n.next, c_c)$;
27. }

28. **Procedure** Merge(T, E, S, c) {
29. **For-each** variable x in T or E
30. Bind x with $c\,?\,v_t \diamond v_f$ into the top table of S
31. **where** v_t is $v(x, T : S)$,
32. v_f is $v(x, E : S)$
33. }

Figure 1. Our symbolic evaluation algorithm

$\mathcal{T}_C$	:=	e	$e \in \mathcal{T}_P$
	|	x_I	$x_I \in$ FPVar
	|	$c\,?\,e_1 \diamond e_2$	$c \in \mathcal{T}_B, e_1, e_2 \in \mathcal{T}_C$
FPVar	:=	Free Preprocessing Variables	

Figure 2. The c-value terms $\mathcal{T}_C$

$\mathcal{T}_P$	:=	$\top$	
	|	$\bot$	
	|	$t_1 \ldots t_n$	$n > 0, t_i \in$ VTok
VTok	:=	Valid Tokens	

Figure 3. The preprocessing (concrete) terms $\mathcal{T}_P$

$\mathcal{T}_B$	:=	b	$b \in$ BVal
	|	$\texttt{def}(x_I)$	$x_I \in$ FPVar
	|	$\neg e$	$e \in \mathcal{T}_B$
	|	$e_1\ o\ e_2$	$e_1, e_2 \in \mathcal{T}_B$, $o \in \{\wedge, \vee\}$
	|	$\bar{\mathrm{Z}}(e)$	$e \in \mathcal{T}_E$
	|	r	$r \in$ Err
def	:	CVal $\rightarrow$ BVal	
$\bar{\mathrm{Z}}$	:	ECst $\rightarrow$ BVal $\cup$ Err	
CVAL	:=	$\{\top, \bot, t_1 \ldots t_n\}$	
BVal	:=	$\{\texttt{true}, \texttt{false}\}$	
ECst	:=	Constants	
FPVar	:=	Free Preprocessing Variables	
Err	:=	Errors	

Figure 4. The final Boolean terms $\mathcal{T}_B$

$\mathcal{T}_E$	:=	c	$c \in$ ECst
	|	x_I	$x_I \in$ FPVar
	|	$o\ e$	$e \in \mathcal{T}_E, o \in$ EUOp
	|	$e_1\ o\ e_2$	$e_1, e_2 \in \mathcal{T}_E, o \in$ EBOp
EBOp	:=	$\{+, <, \#\#, \ldots\}$	
EUOp	:=	$\{\#, \ldots\}$	
ECst	:=	Constants	
FPVar	:=	Free Preprocessing Variables	

Figure 5. The arithmetic terms $\mathcal{T}_E$

$\mathcal{T}_{C\uparrow}$	:=	e	$e \in \mathcal{T}_B$
	|	$\bar{\mathrm{Z}}(e)$	$e \in \mathcal{T}_{C\uparrow}$
	|	$c\,?\,e_1 \diamond e_2$	$c \in \mathcal{T}_B, e_1, e_2 \in \mathcal{T}_{C\uparrow}$

Figure 6. The c-value terms $\mathcal{T}_{C\uparrow}$

evaluation. The complete interpretation of such errors is outside the scope of this paper. It actually depends on the tool using symbolic evaluation. But one simple application is to detect erroneous conditionals and report them to the user. It can also be used to detect erroneous combinations of preprocessing variables on conditionals. Such an example is given in Section 6.

The function $\bar{\mathrm{Z}}$ converts a constant value to `true` or `false`: it gives `true` if the value is a number and not 0. For a non numeric value it gives an error.

The arithmetic terms, described in Figure 5, may appear in Boolean terms. Their occurrence is mainly due to free preprocessing variables. The formal description $\mathcal{T}_E$ is actually too general, since such a term will always contain at least one free preprocessing variable—all operators are applied between constants.

The parsed and evaluated c-value terms $\mathcal{T}_{C\uparrow}$ are presented in Figure 6. These are never bound to preprocessing variables, but are the result of expansion, parsing and evaluation. This occurs at line 13 in algorithm of Figure 1.

```
1. int main(char **argv[], int argc){
2. #if defined(E)
3. #define S "HELLO"
4. #endif
5.
6. #if defined(S)
7.   printf(S);
8. #endif
9. }
10.
11. gcc -DE f.c
12. gcc -DS='"ALLO"' f.c
13. gcc -DS -D'printf(X)=printf("!")' f.c
```

Figure 7. File `f.c`: Any identifier may be a free preprocessing variable (fpvar)

The $\mathcal{T}_{C\uparrow}$ terms may be interpreted as trees whose leaves are Boolean terms $\mathcal{T}_B$. Indeed, as presented in Section 5, the rewrite system R_f applied over $\mathcal{T}_{C\uparrow}$ flatten out the trees to obtain final Boolean terms.

In the following, c-value may refer to the parsed or unparsed case, the context should make it clear.

3 Free Preprocessing Variables

In this paper, a free preprocessing variable (fpvar) is a possible unbound identifier, as found by our symbolic preprocessing algorithm, in a if-directive conditional. It may be a macro. They are all discovered at line 30 of the symbolic algorithm presented in Figure 1: if x is not found according to $v(x, T : S)$ or $v(x, E : S)$, x becomes a fpvar. It will appear in a c-value without a concrete value, that is, as itself.

For example, in Figure 7, identifier `E` is a fpvar since at line 2, in a if-directive conditional, it may be unbound; likewise for `S` at line 6. Line 11 compiles `f.c` defining `E` and indirectly `S`; whereas line 12 directly defines `S`. In both cases, the compile time definitions gave initial values to fpvars.

We might also consider other identifiers as fpvar, but it is in general impossible, from the source code alone, to detect all *intended* free preprocessing variables. For example, line 13 is another valid compilation where identifier `printf` is intended as a fpvar—a macro of rank one. This is impossible to infer unless further information is given outside the source code[5]. Nevertheless, as defined, the fpvar is sufficient for specifying final Boolean terms.

The initial value of a preprocessing variable x is a concrete value, denoted x_I. It may be specified before concrete preprocessing as shown on lines 11 to 13 in Figure 7. Note that in the definition of c-value terms $\mathcal{T}_C$ and $\mathcal{T}_P$ the domains FPvar and VTok are supposed disjoint. That is, a token `x` may represent the fpvar x but it is made distinct to x_I, since the former can resolve—during parsing—to the symbolic value of x whereas the latter represent itself, that is the value given to x outside the source code. It is an essential part of our work that c-value representation is taking into account parsing which occurs during preprocessing. More on this topic in the next sections.

[5] More on this in the conclusion.

```
1. #define R(x) 2##x
2. #define H(x) R(x)
3.
4. #if H(defined(W)) == 20
5.  /* W is not defined */
6. #else
7.  /* W is defined */
8. #endif
```

Figure 8. `defined` is evaluated during expansion

4 Conditional expansion, parsing, evaluation

Although a detailed formal semantics for ANSI C preprocessing could be useful, it is not required for our purpose. Instead, we present an informal explanation of conditionals, detailed enough to define the interaction between concrete preprocessing and the rewrite systems for c-values.

In concrete ANSI C preprocessing, expansion and evaluation do not take place when a variable is defined; but only when a variable is used. That is, a preprocessing variable definition done with `#define`, possibly with parameters, is simply associated with a sequence of tokens. Expansion occurs on the C code itself and on several directives, in particular the if-directives. Only the if-directive expansion and evaluation is of interest to us.

Actually, when the condition of an if-directive is evaluated, there are possible macro expansions and parsing, mixed with the evaluation of the `defined` operator to '0' or '1', and then a final parsing and evaluation based on the operators `==`, `>`, `+`, etc. If the result is zero, it is interpreted as false; otherwise as true.

From the code presented in Figure 8 we can observe the evaluation of the `defined` operator during expansion. If this code is compiled with `W` defined, the macro call `H(defined(W))` evaluates to `H(1)`, then expands into `R(1)` finally reaching `21` by string catenation: the condition `'21 == 20'` evaluates to 0, which is interpreted as false. We might consider the evaluation of `defined(W)` as its expansion, but token substitution and evaluation are mechanisms that we must differentiate to define a precise interaction of preprocessing with c-values.

The full details of `cpp` macro expansion is outside the scope of this paper. It suffices to know that the rewrite system R_d, presented in the next section, interacts with it only

$$\begin{aligned} PE(c\,?\,e_1 \diamond e_2) &= c\,?\,PE(e_1) \diamond PE(e_2) \\ PE(t_1 \ldots t_n) &= \widehat{PE}(t_1 \ldots t_n) \end{aligned}$$

Figure 9. Symbolic parsing/evaluation based on concrete parsing/evaluation

for the `defined` operator.

Once expansion is done, a term $\mathcal{T}_X$ is obtained (see Figure 12). The rewrite system R_e is applied to it to obtain one term $\mathcal{T}_C$.

The symbolic parsing and evaluating function, denoted PE, is applied after macro expansion on a term $\mathcal{T}_X$ and generates a term $\mathcal{T}_{C\uparrow}$. It is based on a modified concrete one, denoted $\widehat{PE}$ as presented in Figure 9. The modified concrete parsing and evaluation functions must handle fp-vars.

The function $\widehat{PE}$ may generate errors of parsing or evaluation. This is actually part of the domain of $\mathcal{T}_{C\uparrow}$.

The result of PE is wrapped into the function symbol $\bar{\mathrm{Z}}$ then the rewrite system R_f, described in the next section, is applied: a final Boolean term is obtained.

The next section presents the three rewrite systems briefly mentioned.

5 Three rewrite systems

As explained in the previous section, for symbolic preprocessing, the different phases of evaluation of a conditional requires different rewrite systems applied to the c-values. Therefore, we will define three rewrite systems:

R_d is applied for the `defined` operator during expansion;

R_e is applied before parsing for the final evaluation of a conditional;

R_f is applied after the final evaluation of a conditional to obtain a Boolean term $\mathcal{T}_B$.

5.1 System R_d for the `defined` operator

During concrete expansion the argument of the `defined` operator is not expanded. Actually the operator is evaluated to 1 if its argument, a valid identifier, has been defined; to 0 otherwise. These 0 and 1 can be used as nominal values for arithmetic and relational operators applied at the final concrete evaluation. For symbolic preprocessing, the argument of the `defined` operator is also not expanded; but its symbolic value is a c-value which should be rewritten to allow the final evaluation.

The rewrite system R_d, presented in Figure 10, is applied during expansion. The function $\widehat{\texttt{def}}$ represents the concrete

$$R_d = \begin{cases} \widehat{\texttt{def}}(c\,?\,e_1 \diamond e_2) \rightarrow c\,?\,\widehat{\texttt{def}}(e_1) \diamond \widehat{\texttt{def}}(e_2) \\ \widehat{\texttt{def}}(\bot) \rightarrow \texttt{`0'} \\ \widehat{\texttt{def}}(\top) \rightarrow \texttt{`1'} \\ \widehat{\texttt{def}}(t_1 \ldots t_n) \rightarrow \texttt{`1'} \\ \widehat{\texttt{def}}(x_I) \rightarrow \texttt{def}(x_I)\,?\,\texttt{`1'} \diamond \texttt{`0'} \end{cases}$$

Figure 10. Rewrite system for operator `defined`

`defined` operator. Note that $\widehat{\texttt{def}}$ ranges over $\{\texttt{`0'}, \texttt{`1'}\}$, whereas `def` ranges over $\{\texttt{true}, \texttt{false}\}$.

The last rule may appear redundant, as it rewrites $\widehat{\texttt{def}}$ into a c-value using `def`. But, in general, this is required since the `defined` operator actually generates a token (`'0'` or `'1'`) which may be used to its nominal value in a numerical operation (e.g. +) or with other operators (e.g. ##, catenation). In many cases, evaluation and other rewrite systems will simplify $\texttt{def}(x_I)\,?\,\texttt{`1'} \diamond \texttt{`0'}$ to $\texttt{def}(x_I)$.

Proposition 1 *The rewrite system R_d is confluent and terminating.*

Proof For termination, the only unclear case is the first rule since the others rewrite into terms without $\widehat{\texttt{def}}$. If we interpret c-values as finite trees, the first rule reduces the application of $\widehat{\texttt{def}}$ to a term with trees of lower heights. Since the trees are finite, this must terminate. It is confluent since there are no critical pairs among the left hand sides of the rules. □

Therefore, once expansion and R_d have been applied, there are no $\widehat{\texttt{def}}$ left in the sequence of tokens and c-values. Such sequences are formally described by terms $\mathcal{T}_X$ in Figure 12. The next subsection describes the rewrite system to apply over them.

5.2 System R_e is applied before parsing and evaluation

The rewrite system R_e, presented in Figure 11 (on the next page), is applied over $\mathcal{T}_X$ before parsing for the evaluation of a conditional.

During concrete preprocessing, an if-conditional is expanded then parsed into a valid conditional expression for evaluation. But for symbolic preprocessing, the variables are bound to c-values: expansion results into a sequence of c-values and tokens; the concrete parser and evaluator cannot be directly applied to it.

The system R_e operates over the terms $\mathcal{T}_X$. Essentially, R_e moves tokens inward c-values and combines adjacent c-values to obtain combinations of sequences of tokens.

Proposition 2 *The rewrite system R_e is confluent and terminating.*

$$R_e = \begin{cases} (c\,?\,e_1 \diamond e_2)\ t & \rightarrow \quad c\,?\,(e_1\ t) \diamond (e_2\ t) \\ \textbf{where } t \in \text{VTok} \cup \text{FPVar} & \\ t\ (c\,?\,e_1 \diamond e_2) & \rightarrow \quad c\,?\,(t\ e_1) \diamond (t\ e_2) \\ \textbf{where } t \in \text{VTok} \cup \text{FPVar} & \\ (c_1\,?\,e_1 \diamond e_2)(c_2\,?\,e_3 \diamond e_4) & \rightarrow \quad c_1\,?\,(c_2\,?\,(e_1\ e_3) \diamond (e_1\ e_4)) \diamond (c_2\,?\,(e_2\ e_3) \diamond (e_2\ e_4)) \end{cases}$$

Figure 11. R_e **is applied over** $\mathcal{T}_X$ **before parsing and evaluation**

$$\begin{array}{lcll} \mathcal{T}_X & := & e & e \in \mathcal{T}_C \\ & | & e_1\ e_2 & e_1, e_2 \in \mathcal{T}_X \end{array}$$

Figure 12. The terms $\mathcal{T}_X$ **after macro expansion**

Proof Termination is certain since all the rewrite rules reduce the length of terms $\mathcal{T}_X$. It is confluent since there are no critical pairs on the left hand sides of the rules.□

The result of R_e is a single c-value. It contains all possible sequences of tokens, as leaves of this tree, to be parsed and evaluated. Once parsing and evaluation have been applied to it, a single term $\mathcal{T}_{C\uparrow}$ is obtained. It is wrapped into $\bar{\text{Z}}$. The next subsection presents a rewrite system to obtain a single Boolean term $\mathcal{T}_B$ from it.

5.3 System R_f is applied after evaluation

The rewrite system R_f, presented in Figure 13, is applied over $\mathcal{T}_{C\uparrow}$ terms. Actually, several rules are not essential as they could be translated by the last one. The other rules simplify the resulting Boolean terms, in the sense that the former reduce the size of the latter.

Proposition 3 *The rewrite system R_f is terminating.*

Proof Interpreting the terms $\mathcal{T}_{C\uparrow}$ as finite trees: all the rules involving $\bar{\text{Z}}$ either rewrite without $\bar{\text{Z}}$ or rewrite $\bar{\text{Z}}$ to trees of lower heights; similarly for Boolean terms. The rewrite rules of c-values always reduce the number of nodes; therefore it must terminate. □

The system R_f is not confluent as the following example shows: $\texttt{true}\,?\,\bar{\text{Z}}(2) \diamond \bar{\text{Z}}(3)$ may be rewritten as $\bar{\text{Z}}(2)$ using rule 1 or as $\texttt{true} \wedge \bar{\text{Z}}(2) \vee \neg\texttt{true} \wedge \bar{\text{Z}}(3)$ using the last rule. Although, they are equivalent from the point of view of Boolean algebra. We state without proof that modulo Boolean equivalence, R_f is confluent.

We can also make R_f confluent by removing the first eight rules. This would increase the size of final Boolean terms. Or better, separate R_f in three rewrite systems, one for $\bar{\text{Z}}$, one for the first rules and one for the last.

In any case, after application of R_f over a term $\mathcal{T}_{C\uparrow}$ one single Boolean term is obtained. This term is used on line 13 of the symbolic algorithm of Figure 1. By induction we can prove that lines 16 and 21 always call V with a c_c being a Boolean term; which proves that line 8 always annotates a node with a Boolean term $\mathcal{T}_B$.

$$R_f = \begin{cases} \texttt{true}\,?\,e_1 \diamond e_2 & \rightarrow \quad e_1 \\ \texttt{false}\,?\,e_1 \diamond e_2 & \rightarrow \quad e_2 \\ c\,?\,\texttt{true} \diamond \texttt{false} & \rightarrow \quad c \\ c\,?\,\texttt{false} \diamond \texttt{true} & \rightarrow \quad \neg c \\ c\,?\,e \diamond e & \rightarrow \quad e \\ c\,?\,(c\,?\,e_1 \diamond e_2) \diamond e_3 & \rightarrow \quad c\,?\,e_1 \diamond e_3 \\ c\,?\,e_1 \diamond (c\,?\,e_2 \diamond e_3) & \rightarrow \quad c\,?\,e_1 \diamond e_3 \\ \bar{\text{Z}}(n) & \rightarrow \quad \texttt{true} \\ \textbf{where } n \neq 0 & \\ \bar{\text{Z}}(0) & \rightarrow \quad \texttt{false} \\ \bar{\text{Z}}(r) & \rightarrow \quad r \\ \textbf{where } r \in \text{Err} & \\ \bar{\text{Z}}(e_1\ o\ e_2) & \rightarrow \quad \bar{\text{Z}}(e_1)\ o\ \bar{\text{Z}}(e_2) \\ \textbf{where } o \in \{\wedge, \vee\} & \\ \bar{\text{Z}}(c\,?\,e_1 \diamond e_2) & \rightarrow \quad c\,?\,\bar{\text{Z}}(e_1) \diamond \bar{\text{Z}}(e_2) \\ c\,?\,e_1 \diamond e_2 & \rightarrow \quad c \wedge e_1 \vee \neg c \wedge e_2 \\ \textbf{where } e_1, e_2 \in \mathcal{T}_B & \end{cases}$$

Figure 13. R_f **is applied over** $\mathcal{T}_{C\uparrow}$ **after evaluation**

```
1. #if defined(F)
2. # define X
3. #endif
4.
5. #if defined(X) && defined(Y)
6.   printf(X);
7. #endif
```

Figure 14. Example with no numerical values

6 Rewrite Examples

In this section we present applications of the rewrite systems on concrete code segments. In the following discussion, the preprocessing variable values are determined by the symbolic algorithm presented in Figure 1. The presented cases are all simple enough to be manually calculated.

Figure 14 shows a code segment with three fpvars.

1. At line 3, X is bound to $\texttt{def}(F_I)\,?\,\top \diamond X_I$.

```
1. #if defined(F)
2. # define X  20
3. #endif
4.
5. #if X > 10 + Y
6.   printf(X);
7. #endif
```

Figure 15. Example with numerical values

2. During the expansion of the conditional at line 5 the term `defined(X)` is expanded to $\widehat{\texttt{def}}(\texttt{def}(F_I)\,?\top \diamond X_I)$ and rewritten to $\texttt{def}(F_I)\,?\,\texttt{'1'} \diamond (\texttt{def}(X_I)\,?\,\texttt{'1'} \diamond \texttt{'0'})$; the term `defined(Y)` is expanded to $\widehat{\texttt{def}}(Y_I)$ and rewritten to $\texttt{def}(Y_I)\,?\,\texttt{'1'} \diamond \texttt{'0'}$

3. The final expansion is the term $(\texttt{def}(F_I)\,?\,\texttt{'1'} \diamond (\texttt{def}(X_I)\,?\,\texttt{'1'} \diamond \texttt{'0'})) \wedge (\texttt{def}(Y_I)\,?\,\texttt{'1'} \diamond \texttt{'0'})$

4. The parsing and evaluation of PE, wrapped with $\bar{\texttt{Z}}$, gives $\bar{\texttt{Z}}((\texttt{def}(F_I)\,?\,1 \diamond (\texttt{def}(X_I)\,?\,1 \diamond 0)) \wedge (\texttt{def}(Y_I)\,?\,1 \diamond 0))$

5. Applying all the rules for $\bar{\texttt{Z}}$ of R_f we have $(\texttt{def}(F_I)\,?\,\texttt{true} \diamond (\texttt{def}(X_I)\,?\,\texttt{true} \diamond \texttt{false})) \wedge (\texttt{def}(Y_I)\,?\,\texttt{true} \diamond \texttt{false})$; applying rule 3 twice we have $(\texttt{def}(F_I)\,?\,\texttt{true} \diamond \texttt{def}(X_I)) \wedge \texttt{def}(Y_I)$; applying the last rule we finally obtain $(\texttt{def}(F_I) \wedge \texttt{true} \vee \neg\texttt{def}(F_I) \wedge \texttt{def}(X_I)) \wedge \texttt{def}(Y_I)$

This last Boolean term could be simplified in some obvious way to $(\texttt{def}(F_I) \vee \texttt{def}(X_I)) \wedge \texttt{def}(Y_I)$ which is indeed the condition under which line 6 is compiled.

In Figure 15 numerical operations are involved in the conditional of line 5.

1. At line 5 the expansion is the $\mathcal{T}_X$ term $(\texttt{def}(F_I)\,?\,\texttt{'20'} \diamond X_I)$ `'> 10 +'` Y_I

2. RE rewrites it to $(\texttt{def}(F_I)\,?\,\texttt{'20 > 10 +'}\ Y_I \diamond X_I\ \texttt{'> 10 +'}\ Y_I)$

3. Parsing and evaluation, wrapped in $\bar{\texttt{Z}}$, give $\bar{\texttt{Z}}(\texttt{def}(F_I)\,?\,20 > 10 + Y_I \diamond X_I > 10 + Y_I)$

4. RF moves the $\bar{\texttt{Z}}$ symbol into the term and rewrites to $\texttt{def}(F_I) \wedge \bar{\texttt{Z}}(20 > 10 + Y_I) \vee \neg\texttt{def}(F_I) \wedge \bar{\texttt{Z}}(X_I > 10 + Y_I)$

In this last example we can see that the concrete parsing and evaluation function must be modified to handle unevaluated free preprocessing variable.

Figure 16 shows a code segment where the if-directive conditional is not syntactically valid without macro expansion. Indeed, if `M` were defined as '`3`', the if-directive conditional would be '`3 0`' or '`3 4`' which are syntactically invalid conditions; but as it is, the condition is always correct

```
1. #if defined(X)
2. # define M   3 <
3. # define Y   4
4. #else
5. # define M   3 ==
6. # define Y   0
7. #endif
8.
9. #if M Y
10.   x += 2;
11. #else
12.   ++x;
13. #endif
```

Figure 16. R_e handles dubious conditionals

```
1. #define M  3 <
2. #if defined(X)
3. # define Y   == 1
4. #else
5. # define Y   4
6. #endif
7.
8. #if M Y
9.   x = 2;
10. #endif
```

Figure 17. Possible parsing error on `M Y`

under the two possible combinations. The rewrite system R_e handles this correctly:

1. At line 7, the c-value of `M` is $(\texttt{def}(X_I)\,?\,\texttt{'3 <'} \diamond \texttt{'3 =='})$ and the c-value of `Y` is $(\texttt{def}(X_I)\,?\,\texttt{'4'} \diamond \texttt{'0'})$

2. The expansion of the condition of line 9 is the term $(\texttt{def}(X_I)\,?\,\texttt{'3 <'} \diamond \texttt{'3 =='})\ (\texttt{def}(X_I)\,?\,\texttt{'4'} \diamond \texttt{'0'})$

3. R_e gives $\texttt{def}(X_I)\,?\,(\texttt{def}(X_I)\,?\,\texttt{'3 < 4'} \diamond \texttt{'3 < 0'}) \diamond (\texttt{def}(X_I)\,?\,\texttt{'3 == 4'} \diamond \texttt{'3 == 0'})$

4. Parsing and evaluation by PE, with $\bar{\texttt{Z}}$ wrapping, gives $\bar{\texttt{Z}}(\texttt{def}(X_I)\,?\,(\texttt{def}(X_I)\,?\,1 \diamond 0) \diamond (\texttt{def}(X_I)\,?\,0 \diamond 0))$

5. Applying R_f gives $\texttt{def}(X_I)$.

Indeed, the conditional at line 9 depends only on whether or not variable `X` is defined. Consequently, symbolic evaluation has found that line 10 is compiled if `X` is defined.

As a last example, Figure 17 shows a case of possible parsing error. At line 8, the conditional is not syntactically valid when `X` is defined: it is '`3 < == 1`'. But it is valid otherwise.

Here is a step by step evaluation:

1. At line 6, `M` is bound to '`3 <`', `Y` is bound to $\texttt{def}(X_I)\,?\,\texttt{'== 1'} \diamond \texttt{'4'}$

2. At line 8, expansion gives the $\mathcal{T}_X$ term $(\texttt{'3 <'})\ \ (\texttt{def}(X_I)\,?\,\texttt{'== 1'} \diamond \texttt{'4'})$

3. R_e gives $\texttt{def}(X_I)\,?\,\texttt{`3 < == 1'} \diamond \texttt{`3 < 4'}$

4. The parsing and evaluation function PE gives $\texttt{def}(X_I)\,?\,\texttt{error} \diamond 1$; wrapped in $\bar{\mathrm{Z}}$ it is $\bar{\mathrm{Z}}(\texttt{def}(X_I)\,?\,\texttt{error} \diamond 1)$

5. RF gives $(\texttt{def}(X_I) \wedge \texttt{error})(\neg\texttt{def}(X_I) \wedge \texttt{true})$

The interpretation of an error value in a Boolean term is up to the application that uses symbolic evaluation. Different purposes may need different interpretations.

7 Related Work

Somé and Lethbridge [18] work addresses the problem of parsing C code in the presence of conditional compilation for the language Mitel-Pascal. It mainly considers parsing combination of code segments under the control of conditional compilation. Logically incompatible conditionals are determined to avoid parsing combinations of incompatible code segments—it reduces the number of parsing instances. But the determination of logically incompatible conditionals does not take into account expansion and control-flow.

Baxter and Mehlich [2] apply a form of rewriting to conditionals of if-directives over the concrete syntax. Given some preprocessing variable bindings, a set of rewrite rules is applied on the concrete syntax; some if-directive conditionals may be simplified and possibly removed if evaluated to `true` or `false`. That work differs from our approach on several aspects: the rewrite rules are based on the local concrete syntax of directives whereas we apply them taking into account control-flow; the rules are heuristics that catch some subset of possible simplifications; macro expansion is not taken into account; and it is used once a set of bindings for some free variables is known. Our approach solves this problem since simplifying Boolean expressions given some bindings provides a rewriting of if-directive conditionals.

Kullbach and Riediger [13] use folding (hiding) and unfolding (showing) on concrete syntax as a visual aid in understanding source code. The technique was implemented as part of the GUPRO program understanding environment. The authors do not address the problem of conditional macro definitions—control-flow of preprocessing is not addressed.

Favre [6] presents APP, an abstract representation of `cpp` directives. It is a formal approach as the author provides a denotational semantics of APP in his Ph.D. thesis; yet there is no efficient technique presented to symbolically use this semantics.

Krone and Snelting [17, 12] perform a limited control-flow analysis of preprocessing to infer the interrelations between code segments. For preprocessing, many aspects are not addressed: macro expansion, macro evaluation and multiple inclusion of files under different variable definitions. Our approach solves this particular problem since these interrelations can be found by combining the Boolean expressions of segments.

Hu *et al.* [10] address similar problems as ours; but as discussed in [14], the technique used is too inefficient to be applicable in practice—the best time complexity grows exponentially as the number of if-directives.

Dehbonei and Jouvelot [3] use a form of conditional expressions for program analysis. It is applied to interprocedural constant propagation analysis. It was not designed to handle a mixture of parsing and evaluation.

8 Conclusion and Future Work

Conditional values are at the heart of our symbolic evaluation of preprocessing. In essence, the conditional values are used to avoid path analysis, but they become useful only if a set of convergent rewrite rules are applied to interact correctly with macro expansion, parsing and evaluation of conditionals.

We have presented three rewrite systems applicable at different phases of macro expansion and evaluation in the presence of conditional values. These different systems are necessary to correctly handle macro expansion and evaluation semantics as in ANSI C. They rewrite any conditional expression containing conditional values into Boolean expression based on the free variables. Once rewritten, conventional techniques of Boolean expression simplifications can be applied.

We have also discussed the problem of free preprocessing variables identification and the expected limitation of any systems trying to determine them.

It would be useful to design a user language to specify not only the free preprocessing variables, but the range of values they might take. In this manner, further information can be provided to symbolic evaluation, and the tools based on it, resulting in more precise analyses. Such a language could be based on conditional values.

We have built a prototype in Scheme, but a complete implementation of our approach in a common working environment, as in xemacs, would provide a better basic module upon which tools—for the practitioner—could be built.

References

[1] G. J. Badros and D. Notkin. A framework for preprocessor-aware C source code analyses. *Software& Practice and Experience*, 30(8):907–924, 2000.

[2] I. D. Baxter and M. Mehlich. Preprocessor conditional removal by simple partial evaluation. In *Eighth Working Conference on Reverse Engineering (WCRE'01)*, pages 281–290, 2001.

[3] B. Dehbonei and P. Jouvelot. Semantical interprocedural analysis by partial symbolic evaluation. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'92), San Francisco*, pages 14–20, June 1992.

[4] N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, pages 243–320. 1990.

[5] J.-M. Favre. The CPP paradox. In *9th European Workshop on Software Maintenance, Durham (England)*, September 1995.

[6] J.-M. Favre. Preprocessors from an abstract point of view. In *Proc. of the International Conference on Software Maintenance*, pages 329–338. IEEE Computer Society Press, Nov. 1996.

[7] Software in the spotlight: FPP, a new implementation of an old preprocessor. *ACM SIGPLAN Fortran Forum*, 15, August 1996.

[8] A. Garrido and R. Johnson. Challenges of refactoring C programs. In *Proceedings of the international workshop on Principles of software evolution*, pages 6–14. ACM Press, 2002.

[9] M. Harsu. Translation of conditional compilation. *Nordic Journal of Computing*, 6(1), Spring 1999.

[10] Y. Hu, E. Merlo, M. Dagenais, and B. Lagüe. C/C++ conditional compilation analysis using symbolic execution. In *Proceedings of the International Conference on Software Maintenance (ICSM'00)*, 2000.

[11] IEC. *ISO/IEC 1539-3 (1999-02): Information technology — Programming languages — Fortran — Part 3: Conditional compilation*. International Electrotechnical Commission, 3, rue de Varembé, PO Box 131, CH-1211 Geneva 20, Switzerland. Telephone: +41 22 919 02 11. Telefax: +41 22 919 03 00. E-mail: `info@iec.ch`. URL: `http://www.iec.ch`, 1999.

[12] M. Krone and G. Snelting. On the inference of confi guration structures from source code. In *Proceedings of the 16th International Conference on Software Engineering, Sorrento, Italy*, 1994.

[13] B. Kullbach and V. Riediger. Folding: An Approach to Enable Program Understanding of Preprocessed Languages. Fachberichte Informatik 7–2001, Universität Koblenz-Landau, Universität Koblenz-Landau, Institut für Informatik, Rheinau 1, D-56075 Koblenz, 2001.

[14] M. Latendresse. Fast symbolic evaluation of C/C++ preprocessing using conditional values. In *Proceedings of the Seventh European Conference on Software Maintenance and Reengineering (CSMR'03)*, pages 170–179, March 2003.

[15] T. C. Lethbridge and N. Anquetil. Architecture of a Source Code Exploration Tool: A Software Engineering Case Study, Computer Science Technical Report TR-9707, University of Ottawa, Ottawa, Canada, November, 1997.

[16] P. Livadas and D. Small. Understanding code containing preprocessor constructs. In IEEE Third Workshop on Program Comprehension, pages 89–97, Washington, DC, USA, November 14–15, 1994.

[17] G. Snelting. Reengineering of confi gurations based on mathematical concept analysis. *ACM Transactions on Software Engineering and Methodology*, 5:146–189, April 1996.

[18] S. Somé and T. Lethbridge. Parsing minimization when extracting information from code in the presence of conditional compilation. In *Sixth International Workshop on Program Comprehension, Ischia, Italy*, pages 118–125, June 1998.

[19] H. Spencer and G. Collyer. #ifdef considered harmful, or portability experience with C news. In *Summer '92 USENIX*, pages 185–198, June 1992.

[20] M. Vittek. Refactoring browser with preprocessor. In *Proceedings of the Seventh European Conference on Software Maintenance and Reengineering (CSMR'03)*, pages 101–110, March 2003.