UNIVERSITÀ DEGLI STUDI DI CATANIA

SCUOLA SUPERIORE DI CATANIA

Paolo G. Giarrusso

TypeChef: Towards Correct Variability Analysis of Unpreprocessed C Code for Software Product Lines

---

DIPLOMA DI LICENZA

Relatori:
Chiar.mo Prof.
Klaus Ostermann
Chiar.mo Prof. Ing.
Giuseppe Pappalardo

Correlatore:
Dr. Christian Kästner

ANNO ACCADEMICO 2010/2011

A mia madre Rosalia

Abstract

Analyzing static correctness of source code of software product lines implemented through conditional compilation in C is extremely difficult. Standard C parsers process only the output of the C preprocessor, which represents only a single variant of the product line, and thus contains no more any information about variability; analyzing a whole product line seems to require parsing C code without first preprocessing it, a task for which no general algorithm is known to date. Because of this, a long line of research explored heuristics yielding approximate results (Garrido and Johnson, 2005, 2003; Padioleau, 2009) and restrictions of preprocessor usage which allow easier parsing (Baxter and Mehlich, 2001; Adams et al., 2009; McCloskey and Brewer, 2005; Kästner et al., 2009).

Therefore, checking if all variants are syntactically and type-correct nowadays essentially requires checking each variant in isolation, an impossible effort for complex feature models, because the number of variants to check is exponential in the number of features.

In this thesis we discuss TypeChef, an on-going development effort which will be for the first time able to analyze together all variants of a software product line.

Contents

List of figures iv

I Introduction 1

1 Introduction 2

II Background 8

2 The C preprocessor 9

2.1 CPP syntax and semantics 9

2.1.1 Macro definition and expansion 10

2.1.2 Conditional compilation 12

2.1.3 Other constructs 12

2.2 Comprehensibility of unpreprocessed code 13

III TypeChef **17**

3 A design for a partial preprocessor **18**

3.1 Requirements 18

3.1.1 PPC as a partial evaluator 21

3.2 Design 32

3.2.1 Conditional compilation 32

3.2.2 The macro table 34

3.2.3 Macro references 35

4 Boolean formula manipulation **39**

4.1 Motivation 39

4.1.1 The need for simplification 40

4.1.2 Existing approaches 41

4.2 Preliminaries 42

4.2.1 Conversion to conjunction normal form 45

4.2.2 Formula renaming 46

4.3 The design space of in-memory representations 48

4.4 Formula representation 52

4.4.1 Simplification 54

4.4.2 Visiting a DAG and formula renaming 58

4.4.3 An exponential example 61

5 Conclusion and future works **63**

5.1 Parsing 63
5.1.1 Token stream representation 64
5.1.2 Typechecking 65
5.2 Related work 65
5.3 Future works 66
5.4 Conclusion 68

Acknowledgements **69**

Ringraziamenti **70**

Bibliography **77**

List of Figures

2.2.1 Interaction of preprocessor facilities 16

3.1.1 Why Eq. (3.1.1) is unsatisfiable – example 1 25

3.1.2 Why Eq. (3.1.1) is unsatisfiable – example 2 27

4.4.1 Standard simplification rules 55

4.4.2 Advanced simplification rules 56

Part I

Introduction

1 Introduction

Many software systems need to support optional or alternative requirements in different usage scenarios. Developers achieve this through various solutions, including settings configurable at runtime. However, often for performance reasons, one wants to generate a product containing only the *features* of interest, which are combined together at compile-time, excluding the others. Therefore, modern software systems are often developed as *software product lines* (SPLs), which allow building customised versions of a software by selecting which features to enable, subject to constraints like dependencies or incompatibilities between features, encoded by a *feature model* through propositional formulas. Feature selections are also called *software configurations*.

Software product lines are often built by using conditional compilation, which is available for various languages, supported either by the language toolchain itself or through external tools. The C language integrates a preprocessing phase before parsing and compilation, performed by a separate component called the C preprocessor (henceforth CPP). Among the various possibilities, it allows conditionally including or excluding code fragments depending on which macros are defined;

one can thus encode the selection of desired features by defining the corresponding macros for the preprocessing phase, and the preprocessor will therefore generate the appropriate program *variant*, which the compiler proper will subsequently analyze. Variability is thus lost after the preprocessing step.

Checking, among others, syntax and type errors for a software product line would certainly be useful to developers, which nowadays cannot guarantee even the absence of syntax errors in all variants. Using static analysis tools for bug checking can be a further step towards correctness of a software. Additional useful questions about variability include: "Which are the possible expansions of a given macro in a given context?", "What code might be generated by this macro usage?", "In which variants is this code fragment included?", "Is this code type-correct under all variants?", "Is this variable initialized in all variants?". Both humans (Favre, 1997; Spencer and Collyer, 1992) and tools (Hu et al., 2000; Latendresse, 2004; Baxter and Mehlich, 2001) face serious difficulties in answering these questions correctly in non-trivial scenarios.

Performing such analysis at once on all variants is desirable. In many product lines (such as the Linux kernel), hundreds or thousands of features can be enabled or disabled, and for n features up to 2^n variants can exist[1], so that checking each one in isolation would be impractical. We need thus to analyse source code before preprocessing eliminates all variability-related information. However, devising a parser for unpreprocessed code yielding correct results in all cases is widely regarded as

[1]The number can be less if the *feature model* restricts which variants are valid, but is still typically exponential in practice.

impossible, due to several problems in the design of CPP:

- CPP directive direct not only conditional compilation, but also facilities for file inclusion (through #include) and macro (through #define and #undef), which interact together. For instance, conditional compilation allows macro to have alternative definitions; in turn, the result of a conditional compilation directive can depend on the expansion of macros, which in turn might have been conditionally defined, and so on.

- Conditional compilation can express compile-time variability, by selecting different code depending on which features are requested, or dually on the facilities offered by the target machine; however, it is equally used to implement *include guards*, a coding pattern which prevents a C header file from being included multiple times and bringing in scope the same declarations, possibly causing compile-time errors. Tools need to distinguish these uses, even if they are implemented through the same constructs.

- Finally, and most problematically, CPP provides lexical, not syntactic macros (Brabrand and Schwartzbach, 2002), and is mostly oblivious to the language, to the point that it is often used as-is in combination with other languages. Lexical macros manipulate token streams, while syntactic macros are integrated into the parser and work at a higher-level, manipulating and producing abstract syntax trees. On one side, this means that conditional compilation directives and macros can appear in place of any token combination and across non-terminals, as we discuss later in Sec. 5.1; thus, a conventional context-

free grammar would need to accept them at each position of each production.

Furthermore, CPP gives no guarantee that produced source code will be syntactically correct, while the design of syntactic macros allow ensuring guarantees.

Source code analysis would thus be easier if variability were implemented through either syntactic macros or other more disciplined solutions, such as compile-time if statements as offered by D[2], feature-oriented programming (Apel and Kästner, 2009), tool-driven feature mapping (Kästner et al., 2008), and so on. However, the large amounts of software product lines implemented using CPP, not only in C, has motivated various research efforts to overcome the technical challenges. Many efforts impose restrictions on the usage of the preprocessor; however, a previous study (Liebig et al., 2011) showed that a significant percentage of CPP annotations are of the *undisciplined* category and thus are quite problematic to process.

Our key conclusion is that it is possible to perform just a part of preprocessing, such that on the one hand variability information is preserved, while on the other hand the resulting language is simple enough to be parsed, with a suitably extended parsing technology.

TypeChef is our research effort to solve these problems, which consists of the following contributions:

- A library which efficiently manipulates boolean formulas.
- A partial preprocessor which performs file inclusion and macro expansion,

[2] http://www.digitalmars.com/d/2.0/version.html

and performs only a partial evaluation of the conditional compilation directives, so that the output still encodes all possible program variants. It thus separates the variability information relevant for further analyses, both by humans and by the rest of our toolchain, from unrelated informations encoded through CPP.

- A parser combinator framework, based on the Scala Parser Combinator library (Moors et al., 2008), but extended to parse the output of our partial preprocessor: the concern of processing variability information is exclusively located within this framework, and the grammar needs thus not to be altered.

- A complete grammar for the GNU C dialect implemented using our framework, developed extending partial implementations

- A variability-aware type-checker operating on abstract syntax trees produced by the parsing process.

- Real world studies of the applicability of these components to real-world Open Source software product lines, including among others the Linux kernel.

We have a preliminary implementation of all these components, with source code available at `http://github.com/ckaestne/TypeChef`.

This thesis discusses the partial preprocessor and formula manipulation, and our approach to avoid exponential complexity in these phases. We will briefly outline the next steps, but for further details we refer to our future paper (Kästner et al.).

The results in this thesis are based on joint work with Christian Kästner, Tillmann Rendel and Klaus Ostermann. Part of this work, concerning the partial preprocessor, has been published (Kästner, Giarrusso, and Ostermann, 2011); however, here we give a different presentation, reflecting the evolution of the partial preprocessor. The main novel contribution of this thesis is a formal definition of correctness for a partial preprocessor, and a more principled analysis of how to perform partial processing in polynomial time.

Part II

Background

2 The C preprocessor

In this chapter we present the fundamental constructs offered by the C preprocessor language (ISO) and their semantics. We concentrate on what is needed to understand subsequent chapters, yet we need to introduce some details which are not usually of concern for developers. Furthermore, in Sec. 2.2, we discuss the readability of CPP constructs, to later discuss how partial preprocessing can also help program understanding.

2.1 CPP syntax and semantics

The C preprocessor language, unlike the C language, is line-based, i.e. the base syntactical units, which are called preprocessor directives, are not terminated by a separator symbol like the semicolon in C but by a whitespace character, i.e. newline. The source code is first divided into preprocessor tokens. Directive lines have a # character as their first non-whitespace character, followed by a token which identifies the directive name. Thus for instance # include <stdio.h> is a valid #include directive. Lines not containing directives undergo macro replacement as described below. We will not detail other transformations performed on such lines, like for

instance comment removal or automatic concatenation of string literals.

2.1.1 Macro definition and expansion

CPP allows to define macros through #define and to undefine them through #undef: a defined macro is associated to a possibly empty *replacement list*, also called *macro body* or *expansion* of the macro. Note that *expansion* can also refer to the processing step of expanding macros. Once defined, a macro is added to the macro table, and can be removed from this table only through #undef. No macro expansion is performed on the macro body at this time; the body can contain references which will be later expanded, as detailed below. Additional macro definitions can be passed through the command line, and they are added to the macro table before processing the input.

Two types of macro exist: object-like macros, introduced by the syntax:

```
#define macroName  macroBody
```

and function-like macros, defined by the syntax:

```
#define macroName(comma-separated macro arguments) macroBody
```

As the subsequent output text is processed, any identifier token matching a macro currently defined in the macro table is a macro invocation and must be expanded.

1. If it is a function-like macro, each usage must provide actual arguments corresponding to the formal parameters of the declaration; macro invocations in the actual arguments are in turn recursively expanded, using this algorithm,

before being substituted in the macro body. This step goes under the name of *argument pre-scan*.

For object-like macros, the macro body is left unaltered.

2. The macro body, as potentially modified by the previous step, is scanned for macro invocations, and they are in turn expanded, using their current definitions, and not the one valid when the macro was defined. In other words, macro definitions are *dynamically scoped*, unlike variables in most programming languages.

   However, a macro definition cannot be recursive: if the macro name appears in its body, even indirectly, this is called a self-reference, and the corresponding token is marked so that it will not be expanded again during preprocessing.

   For function-like macros, this step will process again the expansion of the arguments; if these involved self-reference, the self-reference will be preserved also during the new macro expansion step, because of the marking. Nested macro invocations allow the same self-reference to be processed multiple times and it is still required that it is not expanded, thus the mark must be actually permanently associated to the macro.

3. The result is finally substituted for the original macro invocation.

Since in both cases macros are not allowed to be recursive, they cannot express iteration.

2.1.2 Conditional compilation

CPP allows expressing conditional compilation through #if, #else, #elif, #endif. These directives express an if-then-else-endif construct which allows removing a code fragment from the output, with conditions expressed through *preprocessor conditional expressions*: together with standard C operators, the defined() operator, hereafter abbreviated as def(); def(MACRO) returns a boolean[1] indicating whether MACRO is present in the macro table, as either an object-like or function-like macro. Additionally, #ifdef macro_name is provided as syntactic sugar for #if defined(macro_name).

Moreover, macro definitions can use the stringification operator, written as #, and the token pasting operator, written as ##. # is a unary operator which can be applied to macro parameters and which produces the literal text of its argument, represented as a string constant; ## can be applied to two tokens and concatenates them to form a single token, if possible. If a macro parameter is passed as an argument to any of these operators, the corresponding actual argument does not undergo argument pre-scan.

2.1.3 Other constructs

CPP supports file inclusion through the #include directive. CPP locates the named file and replaces the directive with the file content, which is in turn recur-

[1] In C, booleans are not a primitive type, but they are encoded through integers; 0 represents false, anything different from 0 represents true, but "boolean" operators, like relational operators and def(), return always 1 to represent true.

sively processed. A file is allowed to include itself again, allowing to express recursion. The maximal inclusion depth must have an implementation-defined limit, and since there are no other ways to express iteration, CPP processing is guaranteed to terminate: therefore, CPP is not a Turing-complete language.

Other directives are provided by the preprocessor, i.e. #error, #warning, #pragma, #line. Their implementation, even in our partial preprocessor, is relatively straightforward, therefore we ignore them in the following discussion to concentrate on the interesting and challenging features.

2.2 Comprehensibility of unpreprocessed code

Various studies (Favre, 1997; Spencer and Collyer, 1992; Pearse and Oman, 1997; Krone and Snelting, 1994) show that understanding the control flow in the presence of conditional compilation can be difficult, and that macros defined unbeknownst to a developer will change the meaning of code in unexpected ways. In particular, many uses of macros do not isolate properly the user of the macro from its implementation details: thus, passing for instance an expression containing side effects to a macro defined by #define square(x) ((x) * (x)) will duplicate these side effects, because of the implementation detail that the parameter is used twice.

However, the interaction between different features can create even greater challenges for program comprehension. For instance, we might want to understand whether the body of an #if directive is output or removed, by reasoning on the possible results of a preprocessor conditional expression; however, the result of the

def() operator depends on all the files already processed, many of which have not been written by the same programmer. Looking at CPP with the concepts of programming language theory, we can identify two well-known problems here. On the one hand, reasoning on CPP directives requires tracking mutable state, in this case the macro table, and it is well-known in the functional programming community that this can be difficult. On the other hand, CPP macros behave always like global variables, because they cannot be local to a module other than a specific file.

Consider the example in Fig. 2.2.1: it show a possible CPP input, containing extreme simplification of realistic coding patterns. In particular, the recursive inclusion of lib.h by itself might seem unrealistic, but *indirect* recursive inclusion are relatively frequent due to long include chains.

To understand whether the macro T is defined in line 10, we need to look for definitions like lines 21 and 24 from a header across all previously processed input lines. That means that we need to analyse also gtk.h and any file which is (even indirectly) included before line 10. Furthermore, it is not clear where macros T_IS_32BIT and T_IS_16BIT could have been defined.

This example also shows the coding pattern known as *include guards*, in lines 14, 15, and 27, which prevents the content of the file from being processed again if its inclusion is requested more than once, for instance by different modules; without the include guards, the same declaration might be output twice by the preprocessor, and this could constitute a syntax error; furthermore, a chain of inclusion might lead to a file unwillingly including itself recursively, terminating only when the limit of inclusion depth is reached.

In conclusion, let us remember that for our overall goals, we are primarily interested in the usage of conditional compilation to select features and implement variability, not in other preprocessor features. Additionally, macro definitions and file inclusion just complicate variability analysis. We will see in next chapter how we approach this problem, from the point of view of tool support.

main.c

```
1  #include "lib.h"
2  #if defined(WITH_GUI)
3  #include "gtk.h"
4  #endif
5  #define NAME foo
6  #if defined(NAME)
7  T NAME() {
8    return 3;
9  }
10 #if defined(T)
11 int main() { ... }
12 #endif
13 #endif
```

lib.h

```
14 #if !defined(_LIB_H)
15 #define _LIB_H
16
17 #include <stdio.h>
18 #include "lib.h"
19 extern int open(...);
20 #if defined(T_IS_32BIT)
21 #define T long
22 #endif
23 #if defined(T_IS_16BIT)
24 #define T short
25 #endif
26
27 #endif
```

Figure 2.2.1: Interaction of preprocessor facilities

Part III

TypeChef

3 A design for a partial preprocessor

In this chapter, we outline the requirements (in Sec. 3.1) and design (in Sec. 3.2) of our partial preprocessor, henceforth called PPC.

3.1 Requirements

As discussed in the previous chapters, we are interested in preserving variability information, while processing away uses of other uninteresting CPP constructs. PPC is intended to perform exactly this processing.

Additionally, we allow removing some variability information, by specifying only an incomplete feature selection to PPC; in this case, the output reflects the feature selection and preserves variability related to other features. This allows reducing the complexity of further analysis by concentrating only on some features, considered more interesting.

For the preprocessor there is no semantic distinction between macros used to en-

code features and other ones which instead indicate are used to adapt, for instance, to low-level portability issues. Depending on external requirements, one might want to use a different definition of feature, and policies about such definitions do not belong to the tool itself, but to the users[1]. Therefore, we will abstract in the definitions of this chapter from the specific application to software product lines.

To this end, we separate conceptually the source code defining the SPL proper, which will be input to the partial preprocessor, from what the project uses to encode a specific feature selection, which might be, e.g., either a set of macro definitions specified on the command-line, or a special header file, which is not considered part of the source code, and which is often generated.

We term *macro environment* the set of macro definitions which are external to source code; thus, in the context of software product lines, the feature selection is encoded through a macro environment. However, we will still refer to source code fully preprocessed with a specific macro environment as a *program variant*.

We intend that both programmers and tools should be able to analyse the output; the output of PPC will be represented internally as an annotated token stream, intended for further analysis; programmers will instead read a textual representation of said token stream. For simplicity, we now restrict our attention to the second kind of output, and thus we regard PPC as a source-to-source processor. We will discuss in Sec. 5.1.1 trade-offs concerning the design of an internal representation.

Additional requirements for PPC output follow:

[1]This is also a general design concept rooted in the Unix tradition; see http://catb.org/~esr/writings/taoup/html/ch01s06.html#id2877777

- Intuitively, we require that the output of PPC is a valid input to CPP. Moreover, it should be still possible to generate a specific program variant from the PPC output, by processing it through CPP with a given macro environment; the output should be the same that CPP would produce on the original input[2]. Furthermore, when supplying PPC with an incomplete feature selection, encoded as a partial *macro environment*, and specifying the rest of the feature selection to CPP, the output should again be the same program variant produced by supplying all macro definitions to CPP at once.

  This correctness criteria is not just of theoretical relevance, as it is crucially used during testing. However, the precise definition is more tricky, and we will return to it and make this requirement more precise in Sec. 3.1.1.

- We additionally require that PPC processes completely #include directives, so that no #include directive appears in the output. Also macro expansion should be performed completely: If a macro has multiple alternative bodies, it should be expanded to conditionally compiled text specifying all possible bodies.

- Furthermore, we want to know under which conditions (i.e. macro definitions) a code fragment would be output by the preprocessor. Ideally, the condition should only depend on the externally specified macro environment, as in the work by Latendresse (2003), and we (Kästner, Giarrusso, and Ostermann, 2011) initially specified the same requirement. Therefore, no #define directive should be needed in PPC output. Furthermore, we even decided not

[2] At least, it should be the same token stream, ignoring e.g. differences in whitespace.

to rely on #else and #elif, and to rely only on #if and #endif, for this simplifies reading the code in the common case that #if, #elif and #else are separated by large chunks of program text.

However, we discovered that fully expanding all macros mentioned in a directive leads to practically relevant exponential worst-case behavior, as we will discuss in Chap. 4, because exponential replication of macro bodies; the only solution in the CPP language requires producing #define directives in the output to name subformulas used repeatedly, so that their value is not duplicated. Boolean formulas are however more expressive and can be used to construct conditions which only depend on the external macro environment and which only consume polynomially bounded space.

- We also require the absence of redundant conditional compilation directives. If the condition of a compilation directive cannot be true in the context of outer conditional compilation directives, i.e., it is not satisfiable, it must be omitted together with its body. Similarly, if the condition is always true in its context, then the conditional compilation directive should be omitted. This requirement ensures, among other things, that include guards are correctly handled.

3.1.1 PPC as a partial evaluator

As mentioned, the PPC output should be a valid CPP input, and after CPP processing, the originally intended output should be produced. From this point of view,

PPC is a customized type of partial evaluator (Jones, 1996) for CPP directives: the values of some macros are known at partial preprocessing time, and they should be completely expanded in the output; some other macros are left as yet unspecified, and therefore the code referencing them should be left unevaluated. In the context of variability analysis, we do not specify a value for macros which represent features.

We require however the ability to indicate to PPC that a macro M is undefined, so that def(M) has to evaluate to 0 (which represents a false boolean value). For PPC, unlike CPP, this is different from not specifying the value of macro M: the latter implies that def(M) must be left unevaluated. In both cases, occurrences of the identifier M will be left unaltered. We describe two use cases of this ability, by referring again the example in Fig. 2.2.1.

1. lib.h contains support for 16 bit machines (enabled by T_IS_16BIT), but a product line using that library might not consider that as a possible feature, because our code requires 32 bit machines anyway. In general, a library often supports many possible operating systems, possibly more than our software product line. To save analysis time, we might thus specify that T_IS_16BIT is always undefined.

2. As discussed, lines 14, 15 and 27 encode the include guard pattern. In this and other patterns, the macro is not intended to be defined by the user, but rather it is a flag used for internal purposes by the program. In this case, it prevents the containing file from being processed multiple times. We can thus safely specify that macro _LIB_H is undefined at program entry.

Next, we introduce some definitions which allows making our specification precise. Let L_{CPP} denote the language of programs in unpreprocessed C, which might contain preprocessor directives; let L_C denote the language of programs in pure C. Let Id denote the set of possible macro names; remember that however this set coincides with the set of possible C identifiers, and in particular, even given an input, it is not possible to distinguish unambiguously which are the macro names it references. Let B denote possible macro values; we distinguish an element $\perp \in B$, which we use to represent that a macro is known to be undefined. We term a partial function $\sigma : Id \to B$ a *macro environment*, and Σ be the set of possible macro environments. We can denote that two macro environments are distinct by writing $\mathrm{Dom}\,\sigma_1 \cap \mathrm{Dom}\,\sigma_2 = \varnothing$, and in this case, remembering that functions are identified with their graph, we write $\sigma_1 \cup \sigma_2$ to denote the merge of the two macro environments, which simply contains all macro definitions from both environments. Finally, let $\sigma_\varnothing$ denote the empty macro environment, that is a partial function $Id \to B$ nowhere defined.

Let the function $cpp : \Sigma \times L_{CPP} \to L_C$ denote the behavior of CPP; let finally $ppc : \Sigma \times L_{CPP} \to L_{CPP}$ denote the behavior of the partial preprocessor. Our first attempt to express a correctness requirement for our partial evaluator, similar to correctness in a partial evaluator (Jones, 1996), is then expressed by the following equation:

$$\mathrm{Dom}\,\sigma_1 \cap \mathrm{Dom}\,\sigma_2 = \varnothing \Rightarrow cpp(\sigma_2, ppc(\sigma_1, p)) = cpp(\sigma_2 \cup \sigma_1, p)$$
$$\forall \sigma_1 \in \Sigma, \sigma_2 \in \Sigma, p \in L_{CPP} \quad (3.1.1)$$

Note that it is crucial that $\mathrm{Dom}\,\sigma_1 \cap \mathrm{Dom}\,\sigma_2 = \varnothing$. A macro can expand to a value containing itself, which will not be further expanded; therefore, preprocessing with the same macro environment is not an idempotent operation. We need thus, for each macro definition, to decide whether to supply it at partial preprocessing time or later. As a bonus, this requirement ensures that $\forall\, \mathsf{MACRO} \in Id\,(\sigma_1\,(\mathsf{MACRO}) = \bot \Rightarrow \mathsf{MACRO} \notin \mathrm{Dom}\,\sigma_2)$. In other words, this requirement ensures that macros specified as undefined during partial preprocessing must be actually undefined later.

However, this specification is still impossible to satisfy: Macro expansion is sensitive to the order in which macros are processed. For instance, consider Fig. 3.1.1: file input.c shows a valid preprocessor input, and file output.i shows the output produced by either CPP or PPC with the empty macro environment $\sigma_\varnothing$. As shown in lines 5 and 7, the macro STRINGIFY expands its argument and passes it to _STRINGIFY1, which transforms it into a string via the stringification operator #.[3] Now let us define a macro environment σ_* undefined on all macro names except B, such that $\sigma_*\,(\mathsf{B}) = 2$. Eq. (3.1.1) says now that

$$cpp(\sigma_*, \mathsf{input.c}) = cpp(\sigma_*, ppc(\sigma_\varnothing, \mathsf{input.c})) = cpp(\sigma_*, \mathsf{output.i})$$

but that is incorrect. On line 7, now that B has been transformed into "B" by PPC, it will not be expanded to "2" by the later CPP run; however, "2" would be the result, when directly executing CPP on the original input with environment σ_*.

[3] As shown in lines 4 and 6, macro _STRINGIFY1, which uses directly #, does not macro-expand its argument, as explained in the previous chapter.

input.c

```
1 #define A 1
2 #define _STRINGIFY1(a) #a
3 #define STRINGIFY(a) _STRINGIFY1(a)
4 _STRINGIFY1(A)
5 STRINGIFY(A)
6 _STRINGIFY1(B)
7 STRINGIFY(B)
```

output.i

```
1
2
3
4 "A"
5 "1"
6 "B"
7 "B"
```

Figure 3.1.1: Why Eq. (3.1.1) is unsatisfiable – example 1

For an additional example, consider Fig. 3.1.2, again containing an input and its PPC output. Here macro A (line 2) is defined in terms of B, which is a self-referential macro. As explained, recursion is not supported to ensure termination, therefore B expands to (1+B). If we regard the output as a PPC output and process it again through CPP, the result will not be altered because the definition of B has been removed from the output. Now, suppose that we move the definition of A from input2.c to the initial macro environment σ_1: output2.i is still the expected output. Now let us defer A, i.e., specify it as part of σ_2, the environment passed to CPP after PPC is run. The output would be output2-wrong.i, because in line 3 A is

expanded to B, but B is not expanded to 1+B. The obvious fix is to specify again the definition of B, either in the output of PPC or in the macro environment; however, this would not work as it would affect not only line 3 but also line 4, incorrectly.

Two key problems show up here: When the user defers the specification of A, PPC neither knows whether A is a macro reference or a source identifier, nor that it will be given a definition referencing B. Moreover, expansion of self-referential macros is not idempotent, for after their body is output the special marking of the identifier token which prevented recursive expansion is lost. Therefore, it seems unlikely that PPC could behave according to the specification in Eq. (3.1.1). If the marking were preserved in some way, by using a different output format, or by mangling the name into a different one, preserving the definition of B would be possible; after CPP has been run, one would need to remove the marking, for instance by demangling the identifier name. Strictly speaking, we would need to relax our definition to allow for this; moreover, this would not solve the first problem we presented.

Therefore, we conclude that Eq. (3.1.1) is too strong and therefore could not be satisfied by any partial preprocessor. Our actual requirements are however much weaker. Remember that we are only interested in variability, that in most cases is encoded by defining/undefining macros and testing them with #ifdef or similar, or by giving them true/false values and testing them through #if.

This observation allows weakening our definition, so that it still ensures correctness while restricting which macro definition can be *deferred*, i.e., specified only after partial preprocessing. This restriction is necessary, as a parser cannot cope with

input2.c

```
1 #define B (1+B)
2 #define A B
3 A
4 B
```

output2.i – expected output

```
1
2
3 (1+B)
4 (1+B)
```

output2-wrong.i – result when defining A *after* PPC runs

```
1
2
3 (B)
4 (1+B)
```

Figure 3.1.2: Why Eq. (3.1.1) is unsatisfiable – example 2

unexpanded macro identifiers which might expand to arbitrary bodies – indeed, the whole point of partial preprocessing is to remove these and other constructs.

Let us consider a macro definition d mapping MACRO_NAME to $v \in B$, a given input file input.c, *initial* macro environments σ_1, and deferred macro environment σ_2 containing d. We want to understand if dererring d is *safe*, i.e., if:

$$cpp(\sigma_2 \setminus \{d\}, ppc(\sigma_1 \cup \{d\}, \text{input.c})) = cpp(\sigma_2, ppc(\sigma_1, \text{input.c})) \tag{3.1.2}$$

where $\sigma_1 \cup \{d\}$ denotes σ_1 extended with the definition d, and analogously $\sigma_2 \setminus \{d\}$

denotes σ_2 without the definition d.[4] Putting stringification aside, passing d to PPC, by adding it to σ_1, ensures that all macro references in its body can be expanded by either PPC or CPP, avoiding the problem described in Fig. 3.1.2, and can therefore produce the same result as standard preprocessing; therefore, the interesting question is if deferring it is safe and thus Eq. (3.1.2) holds.

Claim 3.1.3. *We claim that a sufficient condition for safe deferring is that all the following conditions hold:*

- *MACRO_NAME is only mentioned by #ifdef, #if and #elif directives and/or in the body of a #define directive for a deferrable identifier,*
- *v does not reference any identifiers in* σ_1*,*
- *while processing* $cpp(\sigma_1 \cup \sigma_2, \mathit{input.c})$*, MACRO_NAME is never passed as an argument to another macro (so that it cannot be stringified or pasted with other tokens).*

Under these hypothesis, we say that d is deferrable *and write*

$$\mathrm{defer}(\mathit{input.c}, \mathit{MACRO\_NAME}, v, \sigma_1, \sigma_2)$$

Given this definition, our requirement can be expressed as:

$$(\mathrm{Dom}\,\sigma_1 \cap \mathrm{Dom}\,\sigma_2 = \varnothing) \wedge (\forall\, id \in \mathrm{Dom}\,\sigma_2(\mathrm{defer}(p, id, \sigma_2(id), \sigma_1, \sigma_2))) \Rightarrow$$

$$cpp(\sigma_2, ppc(\sigma_1, p)) = cpp(\sigma_2 \cup \sigma_1, p)$$

$$\forall\, \sigma_1 \in \Sigma, \sigma_2 \in \Sigma, p \in L_{CPP} \quad (3.1.4)$$

[4] These notations are coherent with identifying functions with their graph, as is common in set theory.

If non-deferrable macros are instead deferred, i.e., specified to CPP rather than to PPC, Eq. (3.1.1) might still hold, but that is not a requirement; it is actually likely that the definition of deferrable identifiers could be extended, but it is sufficient for our applications.

We have not formally proved that Eq. (3.1.4) holds; we conjecture it is a corollary of claim 3.1.3, which is again a conjecture. Additionally, setting up a formal proof for the full CPP semantics would amount to a substantial amount of work, while proving it on a subset of the CPP language would not necessarily be helpful: restricting CPP by removing stringification and concatenation gives an interesting sublanguage, but important corner cases as the one discussed in Fig. 3.1.1 would no longer be valid, and the required hypothesis would be weaker. The full semantics of CPP do not add syntactic sugar to the language, they enlarge instead expressive power, and add corner cases like the ones we have shown above. Therefore proofs of interesting properties on subsets of the CPP language are of limited interest, and a full proof is left as future work.

However, we use Eq. (3.1.4) as a testing criteria: We use a set of real-world OpenSource programs to test PPC, and the macro environments considered during testing all comply to the hypotheses given.

Non-deferrable feature macros

We discussed the interaction between deferring a macro value and further processing through CPP; however, it is at least as important to consider further processing by the C compiler. Our definition of *deferrable* macro prevents the identi-

fier from appearing in the program outside preprocessor directives. TypeChef rests on the restriction that variability is encoded through conditional compilation and not through macro expansion, because this restriction is applied in practice in most cases and because it is not possible otherwise to guarantee correct results. We found however a few exceptions, but we argue that they are not a real limitation, because they can be solved without modifying the source code.

When processing a section of the Linux kernel, we found one macro encoded a path and was used in C code; the TypeChef C parser could not know that the macro would anyway expand to a string, could not parse the output.

When features are encoded through boolean values (i.e., numbers interpreted as booleans), they can be tested through either #if MACRO_NAME or a C statement similar to if (MACRO) {...}, intended to be evaluated at compile-time thanks to dead-code detection. GNU coding standards[5], recommend this, because it allows a standard C compiler to check syntax and type-correctness correctness of the excluded code path, partially obviating the lack of tools like *TypeChef*. Of course, this only works for the special case where the declarations needed by the feature-specific code are always available.

The problem for TypeChef is again that the source code is only correct if MACRO is actually a macro and its body is convertible to a truth value, and this information is not specified anywhere.

Both kinds of false positives can be solved if the user of PPC encodes appro-

[5] http://www.gnu.org/prep/standards/html_node/Conditional-Compilation.html#Conditional-Compilation

priately the needed information from the feature model, i.e., that such macros are expected to be strings, numbers, or the like. For the case of a feature macro whose value is interpreted as a boolean, one could request PPC to load an extra header containing code similar to:

```
1 #ifdef EXTERN_FEATURE_FOO
2 #define INTERN_FEATURE_FOO 1
3 #else
4 #define INTERN_FEATURE_FOO 0
5 #endif
```

so that the value of the macro used in the source (here, INTERN_FEATURE_FOO) depends on a feature macro handled by TypeChef (here, EXTERN_FEATURE_FOO).

For the example involving paths, one could similarly specify a feature value. Here any string value would probably work, as it does not affect syntactic or type correctness; the specification could be unconditional, or conditional like in the above example. The advantage in this case is that TypeChef will discover the errors which would be produced by not satisfying the constraint, which might be useful if the analysis is also used to reverse-engineer a feature model or to check its correctness.

Based on the limited examples, one might imagine that the need for user intervention is just a technical limitation of our tool which could be easily solved: in our case, TypeChef could just infer that a certain macro needs to be a string and report that if it was not specified by the user. However, in the more general case, macros can expand to arbitrary token sequences, and a specific class of macro bodies can be required for the input to be correct. A related problem is discussed

Analyzing this suggested the absence of obvious meaningful approaches, and we found only one such problem up to now, which we workarounded manually as described. Therefore, further investigating this corner case would have been wasted effort.

3.2 Design

3.2.1 Conditional compilation

Conditional compilation directives divide the program code (including the header text) in a tree of nested *code regions*, such that the leaves of the tree contain no conditional compilation directives. To each code region we associate a preprocessor conditional expression called the *presence condition* (abbreviated p.c.) of that region, so that code in that region would be output by CPP if and only if the associated p.c. evaluates to true. The first code region has presence condition true, because it is unconditionally processed; for a code fragment of the form:

#if C_1 $body_1$ #elif C_2 $body_2$ ... #elif C_n $body_n$ #else $body_{n+1}$ #endif

located in a context with presence condition pc, code region $body_i$ has p.c. $pc \wedge \bigwedge_{1 \leq j < i} \neg C_j \wedge C_i$, $i = 1..n$, where $C_{n+1} = \mathsf{true}$. It is easy to verify the correctness of this definition.

The set of produced formulas takes $\Omega(1+2+3+\ldots+n+n\cdot|pc|) = \Omega(n^2 + n\cdot|pc|)$; this lower bound can be shown easily by considering the case where $|C_1| = \ldots = |C_n| = 1$. In particular, $pc(body_{n+1})$ has size $\Omega(n+|pc|)$. In this

case, even nesting does not change the asymptotic complexity. However, it is possible to represent the p.c. of the various regions in linear space, by creating formulas $S_2, S_3, \ldots, S_n$, with definitions $S_2 = \neg C_1$, $S_{i+1} = S_i \wedge \neg C_{i+1}$ for $i = 2..n+1$, so that $S_i \equiv \bigwedge_{1 \leq j < i} \neg C_j$, and the presence condition of region $body_i$ becomes $S_i \wedge C_i$ for $i = 2..n+1$. It is important that multiple references to S_i are represented as multiple pointers to the same object – next chapters will discuss details of data representation.

The presence conditions for some code regions might be impossible to satisfy for any feature selection. To correctly process multiple inclusions protected by include guards, it is necessary to exclude such regions from the output, by checking whether the presence condition is satisfiable. If this is not the case, we can avoid processing the content of the region, other than to find the matching #endif directive which concludes it. For greater speed we want to use satisfiability checkers for propositional logic, rather than general constraint solver. Therefore, we need to transform preprocessor conditional formulas into propositional formulas.

In particular, this mechanism allows to process correctly include guards: if a file has been included, its include guard will be defined, and if it is included again, the SAT solver will correctly declare that the condition for processing the file content is unsatisfiable. If a file was included only if condition φ was satisfied, and now it is included again under presence condition ψ, it will end up being included again only if $\psi \wedge \neg\varphi$ is satisfiable.

However, preprocessor conditional expression not only contain booleans, introduced through the def() operator and by logical operators as && or || or !, but also in-

tegers, either implicitly converted to booleans, or compared to one another through relational operators, or on which arithmetic operators or macros are applied. Almost always such results can be evaluated at formula parsing time because all variables are known. For the few exceptions, where software variability is encoded through non-boolean features, we can encode the distinct possibilities as feature macros, similarly to what we described in Sec. 3.1.1.

Up to now, we have not specified how variables are interpreted in presence conditions. In particular, it is possible that a condition refers to variables previously defined or undefined within the source code, rather than specified in the initial macro environment. To specify how this case is handled, we deal now with macro expansion.

3.2.2 The macro table

During partial preprocessing we maintain a dictionary, known as macro table, where macros are associated to a set of possible expansions; to each expansion, we associate the boolean condition under which that definition is valid. Additionally, we cache the *existence condition* under which the macro is defined, even if it is always equivalent to the disjunction of all conditions; this condition will be used to evaluate the def() operator. The macro table is updated as macros are defined and undefined during processing, so that for each macro we know under which condition it should be expanded, and for each associated macro body, under which condition it should be used.

The macro environment provided by the user is used to initialize the table, and

the p.c. true is associated to all the macro bodies and is added as existence condition. If a macro was explicitly undefined, i.e., if it maps to $\perp$ (see Sec. 3.1.1), the macro is associated to no expansion and to the existence conditionfalse. Macros whose value is not specified are not stored in the table, and their existence condition is given by definedEx(MACRO), which represents the unknown definition of the macro which the user could specify at a later processing stage.

When under p.c. pc^* an #undef Id directive is processed, PPC applies the substitution $pc \mapsto pc \wedge \neg pc^*$ to the existence condition of Id and the conditions associated to any existing definitions, to reflect the fact that under condition pc^* they have been overwritten and are no more valid.

When macro Id is defined to have body B under p.c. pc^*, PPC first erases the current definitions as done to process #undef, then it adds the entry (B, pc^*) to the other definitions of Id. Entries whose presence condition has become a contradiction can be removed, and if identical expansions appear with different p.c.s, they can be merged together under condition $pc_1 \vee pc_2$.

3.2.3 Macro references

References to other macros can appear outside of CPP directives, in macro bodies, and in presence conditions, but their handling is quite different in the three cases.

1. Outside of CPP directives, we need to be careful to consider all possible expansions for a macro. Therefore, for each defined macro body under condi-

tion pc_i, we perform macro expansion as usual but we wrap the result inside the directive pair #if pc_i...#endif; the results are then concatenated and output. If the disjunction of all these presence conditions, $pc_* = \bigvee_i pc_i$ is not a tautology, we also produce the unexpanded macro reference under condition $\neg pc_*$. As an optimization, we can consider the current presence condition pc: if $pc \wedge pc_i$ is a contradiction for a certain i, we can avoid processing the corresponding expansion.

2. In macro bodies, reference should not be expanded when they appear in the source, that is, where the macro is defined; as discussed, only when the macro body is actually used for replacement the contained macro references should be expanded with the values currently defined: in other words, such macro references are resolved using *dynamic scoping*.

   Therefore, the partial preprocessor does not distinguish at definition time whether identifiers in macro bodies reference other macros or not. When a macro is expanded, the produced source code can simply be scanned again for further macro references as in the previous case.

3. Macro references in presence conditions, instead, are derived from conditional compilation directives, where they would be resolved by CPP when they appear. To obtain the same semantics, we could substitute each macro occurrence by its current expansions, similarly to what we previously described. However, iterating this process leads to an exponential explosion of the formula size. Instead, we need to defer the substitution, and remember

which set of definitions was in scope when the reference occurred. In other words, in this context macro references are resolved using *static* or *lexical scoping*, and we implement this not through eager substitution but by creating a data structure inspired by closures in the implementation of functional programming languages. In our case, each macro reference is associated with a pointer to the existing entry of the macro table.

We need to distinguish two possibilities: MACRO can be referenced either through def(MACRO) or directly.

- In the first case, vastly the more common, the result of def(MACRO) is simply its existence condition.
- Otherwise, we need to perform macro expansion, to be able to parse the condition. The expansion is performed similarly as macro expansion outside of directives; however, we cannot wrap each expansion inside an #if...#endifpair. Since all expansions are mutually exclusive, we create an if...else if...else chain, and represent it using the C conditional operator: condition ? valueThen : valueElse. To convert the resulting formula to a boolean one, instances of such operators can be lifted upwards in the AST representing the formula; for instance, (a ? b : c) == d is equivalent to a? (b == d) : (c == d). We repeat the lifting until valueThen and valueElse are both (convertible to) booleans, so that instances of this operator can be replaced with condition && valueThen || !condition && valueElse. Note that we replicate program code when lift-

ing if upwards, but that is harmless because it is evaluated immediately; however, we duplicate also condition. If the resulting formula is nested within another ?: operator, also the duplication will nest, and condition will appear 4 times. It is here easy to prove by induction that if conditions are nested n times, the size of the formula becomes exponential in n. Discussion of these issues will follow in next chapter.

4 Boolean formula manipulation

4.1 Motivation

The TypeChef toolchain can partially preprocess, parse and type check a software product line only if it can *efficiently* construct, manipulate and check satisfiability and validity of the boolean formulas produced along the process. By efficient we mean that we can tolerate only algorithms of polynomial space and time complexity, and that sometimes even quadratic complexity can be excessive, depending on the size of the involved parameter.

We already discussed in the previous chapter why checking satisfiability is important during partial preprocessing; in later phases, it becomes even more important. For instance, checking whether an identifier is declared before use entails checking whether the presence condition of its use implies the presence condition of its definition.

Additionally, we need to pretty-print formulas to the user, especially during partial preprocessing: to allow the user to fix effectively the errors we found, it is desirable to simplify formulas before outputting them, since otherwise redundant

clauses (which arise extremely often) make them unreadable.

Finally, it should ideally be possible to determine directly under which cases a section of the output is included. We had among our initial design goals the wish to produce output depending only on the macros externally defined by the feature model – that is, on the *external definitions*.

Naive implementations of these algorithms have exponential complexity, and designing efficient algorithms turned out to be unexpectedly tricky and challenging. In this chapter we analyze the causes of this exponential complexity and discuss how we are optimizing the implementation to scale up to its requirements through known general techniques for symbolic computation, for formula manipulation, and through optimization steps specific to the domain of partial preprocessing.

4.1.1 The need for simplification

Let us consider a small example to see what happens without simplification. Suppose that macro A is defined conditionally:

```
1 #if FEAT1 && FEAT2
2 #define A BODY1
3 #else
4 #define A BODY2
5 #endif
```

Suppose that macro A is used to conditionally define B as follows:

```
1 #if FEAT2
2 #define B A
3 #endif
```

Without any simplification, the expansion of B would become:

```
4 #if FEAT2 && FEAT1 && FEAT2
5 BODY1
6 #endif
7 #if FEAT2 && !(FEAT1 && FEAT2)
8 BODY2
9 #endif
```

In this case, one can easily see that both formulas contain redundant subformulas; the first could be simplified to FEAT2 && FEAT1, and the second to FEAT2 && !FEAT1; it also seems simple for a computer to perform such simplifications. However, the above example is an extremely simple case. In real-world examples, processing a single source file can involve processing thousands of #include directives and hundreds of thousands of lines of code; conditions are more complex and more deeply nested; many inclusions are conditional, so that all the macros that they define are conditionally defined; often, the same headers might be included again under under different conditions.

Simplifying formulas well and quickly in such cases is not simple; on the other hand, it becomes even more important exactly because conditions are more complex to understand. Moreover, simplifying formulas removes repetitions and thus crucially reduces formula size.

4.1.2 Existing approaches

Many standard algorithms for formula manipulation are based on *binary decision diagrams*, or BDDs, in particular *reduced ordered* BDDs (Bryant, 1986), and

they are also used to manipulate boolean formulas in variability analysis (Mendonça et al., 2008). Constructing BDDs requires choosing a fixed variable order, but for some choices the size of a BDD suffers from *combinatorial explosion*, i.e., it becomes exponential; determining the optimal variable order is an NP-hard problem. Using existing heuristics, SAT-based approaches have successfully handled reasoning on up to 10 000 features (Mendonça et al., 2009), while with BDDs late results reach only up to 2 000 features (Mendonça et al., 2008). For a comparison, the Linux kernel provides 8 000 features, and this motivated our choice of a SAT-based approach.

Even though checking satisfiability of a propositional formula, or SAT-solving, is notoriously an NP-complete problem, commonly referred as SAT, modern SAT solvers are quite efficient on many classes of instances; formulas arising in tools for SPL support, in particular, are always tractable (Mendonça et al., 2009), and testing of our tools never produce intractable formulas during development.

However, the problems that we mentioned arise in manipulation steps other than SAT-solving: when naively implemented, these algorithms consume exponential space and/or time in the worst case; this is not only a theoretical concern, but it actually makes analysis of some real-world C programs intractable.

4.2 Preliminaries

In this section we introduce some standard definitions for propositional logic that we will later need.

To define propositional formulas, we must first stipulate a set of variables, and we choose the set of macro identifiers. In this chapter, unless otherwise stated, variables (e.g., MACRO) represent the result of the def() operator (e.g., def(MACRO)).

A *literal* is an atom (a) or the negation of an atom ($\neg a$).

A *formula* (φ) is either an atom (a), a negation of a formula ($\neg\varphi$), a conjunction of two formulas ($\varphi \wedge \psi$), a disjunction of two formulas ($\varphi \vee \psi$); evaluation is defined according to the usual semantics.

Since $\wedge$ and $\vee$ are commutative and associative, we can unambiguously extend them to sets: if S is a set and $s_1, s_2, \ldots, s_n \in S$, we define $\bigwedge_{s \in S} = s_1 \wedge s_2 \wedge \ldots \wedge s_n$ and $\bigvee_{s \in S} = s_1 \vee s_2 \vee \ldots \vee s_n$.

We call a disjunction of literals which does not contain opposite literals a *clause*; a formula in *conjunctive normal form* (henceforth CNF) is a conjunction of clauses. The conjunctive normal form of a formula is important because SAT solvers require as input a formula in this form.

A first step into converting a formula to conjunctive normal form is conversion to *negation normal form* (NNF). A formula is in NNF if and only if the $\neg$ operator is only used on atoms. In other words, negation cannot be used on the result of other operations, but it must be "pushed down" as far as possible. Any propositional formula can be converted to NNF by applying repeatedly the following rules:

$$\neg\neg\varphi \mapsto \varphi$$

$$\neg(\varphi \wedge \psi) \mapsto \neg\varphi \vee \neg\psi$$

$$\neg(\varphi \vee \psi) \mapsto \neg\varphi \wedge \neg\psi$$

Each of these rules decreases the size of negated formulas, guaranteeing termination of the algorithm; when the algorithm terminates, no pattern applies to the resulting formula. By inspecting the rules, one can convince himself that the result must therefore be in NNF.

Through the above defined operators, one can define the additional *derived* operators $a \to b = (\neg a \vee b)$ and $a \leftrightarrow b = (a \to b) \wedge (b \to a)$; formulas using those operators are immediately converted in their simplified form which uses only the primitive operations $\neg, \wedge, \vee$.

An assignment is a partial function mapping variable names to booleans, defined only on a finite number of input values; an assignment A *models* a formula φ, written $A \models \varphi$, if A defines all the variables mentioned φ so that the latter evaluates to true.

A formula φ:

- is *valid*, or a *tautology*, if it all assignments model it: We write then $\models \varphi$, or $\varphi = \top$, where $\top$ represents true, or can be taken to be any specific tautology;
- is *unsatisfiable*, or a *contradiction*, if its negation is valid, i.e. $\models \neg\varphi$; we write then $\varphi = \bot$, where $\bot$ represents false, or can be taken to be any specific contradiction;
- is *satisfiable* if it is not unsatisfiable $\not\models \neg\varphi$; Equivalently, φ is satisfiable iff there is an assignment A that models it, which we write $\exists A\ (A \models \varphi)$, with a slight notational abuse.

Two formulas φ and ψ are *equivalent* and we write $\varphi \equiv \psi$ iff for any assignment A, $A \models \varphi \leftrightarrow A \models \psi$; they are *equisatisfiable* if they are both satisfiable or both

unsatisfiable, under possibly *different* models.

Finally, a formula transformation f preserves satisfiability if $f(\psi)$ is equisatisfiable to ψ for any ψ, and preserves equivalence if $f(\psi) \equiv \psi$ for any ψ.

4.2.1 Conversion to conjunction normal form

A generic formula can be converted to CNF by first converting it into NNF and then applying repeatedly the distributive law of disjunction over conjunction: $(x_1 \wedge y_1) \vee (x_2 \wedge y_2) = (x_1 \vee x_2) \wedge (x_1 \vee y_2) \wedge (y_1 \vee x_2) \wedge (y_1 \vee y_2)$.

From the point of view of the implementation, if the formula is implemented as a tree, the CNF can be defined recursively after transforming it into NNF: each node computes the CNF of its children and combines them appropriately. In particular, disjunction nodes need to apply the distributive law; conjunction nodes can simply join together the CNF formulas obtained from their children; negation nodes in NNF are only part of literals, thus they are already in CNF. For efficiency, the conversion to NNF and the later pass of conversion to CNF can be fused together in a single-pass algorithm, but we omit further details because mostly straightforward.

However, when a formula is a disjunction of n conjunctions of literals $c_1, c_2, ...c_n$, applying the distributive law produces $\prod_{i=1}^{n} c_i$ clauses. We show an example for

$c_1 = c_2 = \ldots = c_n = 2$:

$$\begin{aligned}\bigvee_{i=1}^{n}(x_i \wedge y_i) = {} & (x_1 \vee \ldots \vee x_{n-2} \vee x_{n-1} \vee x_n) \\ & \wedge (x_1 \vee \ldots \vee x_{n-2} \vee x_{n-1} \vee y_n) \\ & \wedge (x_1 \vee \ldots \vee x_{n-2} \vee y_{n-1} \vee x_n) \\ & \wedge (x_1 \vee \ldots \vee x_{n-2} \vee y_{n-1} \vee y_n) \\ & \wedge \ \ldots \\ & \wedge (y_1 \vee \ldots \vee y_{n-1} \vee y_n)\end{aligned} \tag{4.2.1}$$

We will later present the general technique of *formula renaming*: renaming a formula preserves satisfiability and allows producing an equisatisfiable formula in CNF having size polynomial in the input. Furthermore, this technique will be useful for us to avoid combinatorial explosion also in other algorithms for formula manipulation.

4.2.2 Formula renaming

Let us first illustrate the technique on the same example. We need to transform $\chi = \varphi \vee \psi$ into CNF; we assume that both φ and ψ have been transformed into CNF and have respectively n and m clauses, but we want to avoid producing nm clauses. We now introduce two fresh variables x and y and consider the formula $\chi' = (x \vee y) \wedge (x \leftrightarrow \varphi) \wedge (y \leftrightarrow \psi)$; in this formula, we have *renamed* φ and ψ respectively by x and y, producing the clause $x \vee y$, and we have conjuncted formulas ensuring that (in a satisfying assignment) each formula evaluates to the same truth value as its "new name". Indeed, one can prove that χ and χ' are equisatisfiable.

In general, we can rename an arbitrary subformula φ of χ; we have to be careful, however, to add the clauses relating φ to its new name only to the top level.

The technique, as described, allows renaming an arbitrary subformula, but is not always optimal; renaming φ adds the CNF of $x \leftrightarrow \varphi = (x \rightarrow \varphi) \wedge (\varphi \rightarrow x) = (\neg x \vee \varphi) \wedge (\neg \varphi \vee x)$, which adds two occurrences of φ in place of the renamed one. However, this doubling happens exactly once and does not lead to a exponential blowup, because the resulting clauses are added to the top-level, rather than inside the formula where they could be subject to renaming and be duplicated again.

Still, when using formula renaming under special conditions, it is possible to avoid this duplication. We avoid stating those conditions in full generality, and refer the reader to Nonnengart and Weidenbach (2001) for those. We simply note that when χ is in NNF and we rename any subformula φ which is not a literal, and thus does not appear inside any negation, it is enough to add $x \rightarrow \varphi$ rather than $x \leftrightarrow \varphi$; we allow x to be false while φ is true, but this "loss of precision" is not a problem, because after modifying the assignment to make x true produces an assignment which still models χ, thanks to the constraints on χ and φ.

What is even more useful is that these hypothesis apply when converting a formula from NNF to CNF. Therefore, the original example $\chi = \varphi \vee \psi$ is equisatisfiable to $\chi' = (x \vee y) \wedge (x \rightarrow \varphi) \wedge (y \rightarrow \psi)$.

4.3 The design space of in-memory representations

It is possible to translate directly the above definitions to a BNF grammar, and to use the associated AST as our data structure.[1] Formally, in BNF notation:

$$\begin{aligned} Formula ::= & \; Identifier \\ & \mid Formula \wedge Formula \\ & \mid Formula \vee Formula \\ & \mid \neg Formula \end{aligned} \tag{4.3.1}$$

To represent $\neg a$, $a \vee b$, $a \wedge b$, we simply build a new AST node.

In all cases, both for this representation and the next ones we will discuss, instances of all these classes are (semantically) immutable once constructed: We cannot modify a formula, rather we create a new one. Due to extensive sharing, this is essential: otherwise we might risk modifying, e.g., the presence condition of already created nodes.

The problems with such a representation are the memory consumption and simplification. For instance, $n_1 = \neg(\neg(\neg a))$ is a valid node, as $n_2 = (((a \wedge b) \wedge c) \wedge a)$, or as $n_3 = (a \wedge b) \wedge (c \wedge a)$. The size of a formula grows at each operation, and fully simplifying a formula is expensive, because it requires looking for duplicate subformulas, and comparing two formulas for structural equality has a cost linear in their size. Thus, in a tree with n nodes, comparing each pair of nodes could take (by a very simple estimation) $\Omega(n^2)$), which is quite expensive. Additionally, structural

[1] Readers knowledgeable with functional programming will notice that we are defining in essence an algebraic data type, as available for instance in Haskell, even if with a more sloppy notation.

equality cannot detect that, e.g., $n_1 = \neg a$, or $n_2 = n_3$, even if those formulas are clearly equivalent, and differ only in the order of their operation.

We described just the simplest possible algorithms based on this data structure; among other things, one can try simplifying just some common patterns for which simplification is less expensive, or disable it altogether. However, we experimented with various variations, but simplification had never good performance, and even disabling it altogether, conversion of formulas into CNF always consumed excessive time and space. Thus, in the end we abandoned this approach, and we found one which is much better, that we later describe.

One way to look at the problem is that the data representation can represent too much information which is irrelevant, for instance the difference between n_2 and n_3. Analogously, there is no reason to allow building $n_4 = \neg(\neg a)$: when attempting to construct such a node, a good algorithm should simply return a. Since $n_1 = \neg n_4$, attempting to build n_1 would simply produce $\neg a$ again. This suggests using a *canonical* or *normal* form, such that a single representation is used for both n_2 and n_3, and that we prevent building nodes like n_4 *during construction*.

A canonical form on a set S is always relative to some equivalence relation $R \subset S \times S$, because the canonical form of an element $x \in S$ is a designated element of its equivalence class. Now, it is crucial to understand that we cannot use a representation which is canonical relative to formula equivalence, but we need to use a smaller relation. Otherwise, either the canonicalization algorithm or checking structural equivalence on the result would be NP-hard, as can be easily proved:

Theorem 4.3.2. *Converting any propositional formula (represented as defined by*

Eq. (4.3.1)) to a form which is canonical relative to equivalence, and which can be compared for equality in polynomial time, is NP-hard.

We present a proof sketch.

Proof. Let us assume that $c(\psi)$ computes a canonical form of ψ in polynomial time for a generic ψ. We can check whether a formula φ is valid by comparing its normal form $c(\varphi)$ with $c(\top)$, where $\top$ is an arbitrary tautology; since c has only time to produce an output of polynomial size, the comparison would take polynomial time as well. We can check satisfiability with a validity checker: $\exists A\ (A \models a) \leftrightarrow \not\models \neg a$. Since in the representation above $\neg a$ can be produced in polynomial time, this gives an polynomial algorithm for SAT. In technical terms, we have reduced SAT to the canonicalization algorithm c, and thus shown that the latter solves an NP-hard problem. □

BDDs are such a canonical form, and this theorem explains why they suffer exactly from this problem. The same is true for some CNF variants, but conversion to CNF (also to such variants) produces output of exponential size in the worst case. Converting a formula to an equisatisfiable one in CNF form, on the other hand, introduces freshly generated variables in the formula, so that equality would need to be defined up to renaming of those variables.

However, we are not really interested in formula equivalence, as the SAT-solver takes care of that; our motivation is increasing readability of produced formulas while enabling simplification patterns to remove redundant subformulas.

A canonical form has the following advantages:

1. Structural equality between object in a canonical form allows testing the relation R.

2. If we can construct an hash function consistent with structural equality, and our data is represented through immutable objects, we can avoid having multiple copies of the same object, ensuring *maximal sharing*.

3. Maximal sharing not only saves memory, but allows testing for structural equality through a simple pointer comparison; this changes the comparison cost from $\Omega(n)$ to $\Theta(1)$. This in turns is useful for fast simplification.

4. It simplifies code operating on the structure, because it needs to cope with a smaller variety of structures.

Similar technique have long been used since long time, especially in symbolic computation, and go under different names: maximal sharing, hash-consing (Goto, 1974; Ershov, 1958; Appel and Gonçalves, 1993), and the flyweight pattern (Gamma et al., 1995). A modern description of hash-consing, together with an experimental evaluation of it applied to boolean formula manipulation, is presented by Filliâtre and Conchon (2006).

We denote structural equality of formulas through $=$, and pointer equality by $\stackrel{\circ}{=}$. Thus, $\varphi \stackrel{\circ}{=} \psi$ means that φ and ψ are represented by the same object. In general, $\varphi \stackrel{\circ}{=} \psi \rightarrow \varphi = \psi$; with maximal sharing, $\varphi \stackrel{\circ}{=} \psi \leftrightarrow \varphi = \psi$.

4.4 Formula representation

In our representation, implemented in Scala[2], we have a simple object hierarchy to represent formulas. FeatureExpr is the root of our hierarchy; subclass DefinedExpr represents atoms; subclass Not represents the negation operator, and contains a field referring to the negated formula; $\wedge$ and $\vee$ are represented by subclasses And and Or, both inheriting from BinaryLogicConnective. They both contain a field which links to a Set of formulas; Set are implemented by Scala standard library through hashtables, thus membership testing takes time constant in the set size; for the time to be constant also in the formula size, we need $O(1)$ equality testing and hash-code computation. Finally, we have singleton objects True and False; since True is the identity for $\wedge$, we stipulate that And(Set())[3] is identical to True, so that True extends And(Set()). Dually, Or(Set()) is identical to False. [4]

All formulas inherit the default implementation of structural equality (method Object.equals), which only tests for pointer equality, and has thus $O(1)$ cost. Furthermore, each subclass of FeatureExpr has a amortized $O(1)$ hash-code implementation. To ensure this, BinaryLogicConnective contains a cache of the hash code, to ensure that it is computed only once. Furthermore, given a = And(Set(x_1, x_2, ...,

[2]Scala is a modern programming language running on the JVM and with strong compatibility with Java libraries, which is both object-oriented and functional; we assume only knowledge of Java in the discussion.

[3]Set() represents the empty set.

[4]This was inspired by the algorithm of *resolution*, where a contradiction is represented by the empty clause. However, if we define $\bigwedge$ and $\bigvee$ over sets of clauses, the only coherent way to extend them to $\varnothing$ is the one we described, i.e., $\bigwedge \varnothing = \top$, and $\bigvee \varnothing = \bot$.

x_n)), creating b = And(Set(x_1, x_2, ..., x_{n+1}) reuses the old hash code for computing the new one, in this case at object time.

The constructors of all these classes are hidden, and factory methods must be used: this is needed to ensure that no two objects representing the same formula are created, and that formulas are simplified during object creation.

After a new object is created, we need to check whether another copy of it was already created. The standard solution is to look up an object in a canonicalization hash-table, where we associate to each canonical object the object itself; thus, looking up in this table an object o' equal to the already existing o returns o, which we can return. However, during this lookup we cannot use pointer equality to implement structural equality, because we are trying to detect equal formulas to eliminate the duplicate from existence. The key idea of hash-consing is that since object instances are canonicalized at build time, it is correct to use pointer equality for the children of the current node. Thus, for an instance of BinaryLogicConnective having k children, we can check equality in $\Theta(k)$ time, by checking whether each child of the newly created node belongs to the set of children of the existing node, and whether the two sets of children have the same size.

We can safely skip this step when enlarging an BinaryLogicConnective node with a new operand, if the node is used internally for adding further clauses, as long as we finally canonicalize the result.

4.4.1 Simplification

As a first step, we notice that Not(Not(a) ≡ a, And(And(a, b), c) ≡ And(a, b, c), Or(Or(a, b), c) ≡ Or(a, b, c). Therefore, we enforce as a structural constraint that connectives of the same type cannot be nested: whenever the library is requested to build one, it builds the flattened version, similarly to the examples just described.

We use the transformation rules in Fig. 4.4.1 and Fig. 4.4.2 for formula simplification. Thanks to the use of *Set*, they are matched commutatively.

The two tricky pattern are quite useful: they ensure that in a (sub)formula of shape $\varphi \curlyvee (\curlywedge_i \psi_i)$, where $\curlyvee$ and $\curlywedge$ are any two binary operators, neither φ nor $\neg\varphi$ appear among $\{\psi_i\}_i$, because they will be simplified.

Adding DeMorgan's laws was particularly tricky, for various reason, but also particularly useful: they ensure that formulas are normalized to NNF while construction. A complexity analysis shows that applying De Morgan laws during formula construction, i.e. bottom-up, can cause a subtree to be processed each time the containing tree is negated. Let us consider iterating the transformation, $a \mapsto \neg(a \wedge b)$, where b is a fresh atom (i.e., not used in a), so that $a \wedge b$ is not simplified by any pattern, and the result of applying it n times. At each step the whole obtained formula is visited again, leading to complexity quadratic in n and in the input size if the tree size is linear in the number of step (i.e., if it is list-like). In comparison, if we perform the transformation top-down, we can stop building new nodes when we find a subformula of form Not(f); we simply obtain f as its negation; afterwards, we continue descending looking for new negated formulas. Therefore,

$$\neg True \mapsto False \tag{4.4.1}$$

$$\neg False \mapsto True \tag{4.4.2}$$

$$e \wedge e \mapsto e \tag{4.4.3}$$

$$e \wedge False \mapsto False \tag{4.4.4}$$

$$e \wedge True \mapsto e \tag{4.4.5}$$

$$e \wedge \neg e \mapsto False \tag{4.4.6}$$

$$e \wedge (e \wedge e') \mapsto e \wedge e' \tag{4.4.7}$$

$$e \wedge (\neg e \wedge o) \mapsto False \tag{4.4.8}$$

Duals:

$$e \vee e \mapsto e \tag{4.4.9}$$

$$e \vee True \mapsto True \tag{4.4.10}$$

$$e \vee False \mapsto e \tag{4.4.11}$$

$$e \vee \neg e \mapsto True \tag{4.4.12}$$

$$e \vee (e \vee e') \mapsto e \vee e' \tag{4.4.13}$$

$$e \vee (\neg e \vee o) \mapsto True \tag{4.4.14}$$

Figure 4.4.1: Standard simplification rules

Tricky rules:

$$e \wedge (e \vee o) \mapsto e \tag{4.4.15}$$

$$e \wedge (\neg e \vee o) \mapsto e \wedge o \tag{4.4.16}$$

Duals:

$$e \vee (e \wedge o) \mapsto e \tag{4.4.17}$$

$$e \vee (\neg e \wedge o) \mapsto e \vee o \tag{4.4.18}$$

De Morgan laws:

$$\neg \bigwedge_i x_i \mapsto \bigvee_i \neg x_i \tag{4.4.19}$$

$$\neg \bigvee_i x_i \mapsto \bigwedge_i \neg x_i \tag{4.4.20}$$

Figure 4.4.2: Advanced simplification rules

the whole formula is only visited once.

However, this problem can be solved effectively by memoizing formula negation. We first add a field notCache to the class FeatureExpr; when we negate a formula φ and produce $\psi \equiv \varphi$, we set φ.notCache = ψ and ψ.notCache = φ, so that next time we negate either formula we can return the result immediately. Note that maximal sharing makes memoization more effective and cheap. Alert readers might notice that adding such a cache seems to make objects no more immutable; however, the *meaning* of each node is still immutable, and this is the essential for the correctness of our software. If a formula is the presence condition of multiple statements, we cannot add a clause because that would change the p.c. of those statements, but caching its negation within that formula is not a problem.[5]

This still does not solve the problem; with this fix, negating $(\bigwedge_{i=1}^{n} \psi_i)$ takes $O(n)$ time, if for each ψ_i the negation was already computed, and time proportional to the formula size in the worst case. However, applying rules like (4.4.8) relies on negation, which must then take constant time to be fast. In practice, this change makes runtime again exponential; to fix the problem, we changed these rules, so that they are applied only if the needed negations have already been computed. The underlying hypothesis is that if objects representing ψ and $\neg\psi$ exist in memory, it is likely that $\neg\psi$ has been produced by negating ψ, and that therefore the cache lookup will be successful; otherwise, if the cache lookup fails, it is likely that the

[5]Immutable data is also useful because it prevents data races in multithreaded software and, in that context, material immutability is important. However our partial preprocessor is single-threaded, so this is not a problem.

negation has never been computed, and therefore it is useless to build $\neg\psi$ to use it in a comparison which is likely to fail. While we might miss some opportunities for simplification, the runtime becomes again acceptable, we can apply DeMorgan's laws during construction and represent formulas in NNF, and this enables other simplification opportunities. For instance, we observed in the output of our tool occurrences of the pattern $a \wedge \neg(a \wedge b)$, and its variation; in NNF this formula becomes $a \wedge (\neg a \vee \neg b)$, which is simplified by rule (4.4.16) to $a \wedge \neg b$. However, we will need to perform more benchmarks to understand whether NNF normalization is worth its cost.

Another effect of these patterns is that the associative property does not hold any more wrt. structural equality, because of simplification patterns which trigger in one case but not another. For instance the associative law for $\wedge$, i.e. $((a \wedge b) \wedge c) \not\equiv ((a \wedge b) \wedge c)$, does not hold for $(a,b,c) = (x, x \vee y, z)$, because $a \wedge b = x \wedge (x \vee y) \mapsto x$.

4.4.2 Visiting a DAG and formula renaming

Converting a formula to CNF, or applying DeMorgan's laws to a formula, is done by a recursive visit, i.e., in essence, through a post-order visit of the tree representing the formula. However in our cases subtrees can be shared, first because they are immutable so they might be shared, second because we have maximal sharing, so wherever sharing is possible it will happen. This means that our data structure, under an appearance similar to an abstract syntax tree, is actually a disguised directed acyclic graph, which requires the use of different algorithms to avoid visiting the same node more than once.

On the other hand, if we now want to output the formula, we seemingly need to visit the same trees again.

Additionally, maximal sharing can change the size of a data structure from exponential to polynomial, therefore visiting again node can make the complexity of the visit exponential.

For example, consider the family of formulas (and the representing DAGs) defined as: $T_1 = a$, $T_{i+1} = T_i \wedge (b \vee (T_i \wedge \neg c))$. This complex family is chosen to avoid that simplification patterns simplify it and remove the duplication. Taking sharing into account, the cost of representing the DAGs $T_1, \ldots, T_n$ is $\Theta(n)$, because the representation of T_{i+1} given T_i takes $O(1)$ space. However, if we regard them as trees, their size is exponential. Using the number of leaves as a size measure, $|T_{i+1}| = 2\,|T_i| + 2$, so that the number of leaves for T_n is $\Omega(2^n)$; this implies that also the number of nodes in the corresponding tree is exponential.

In standard DAG algorithms, one maintains a set of visited nodes, or flags the nodes themselves; however, this complicates the traversal. In the examples we considered, memoization however is a convenient way to mark nodes which have already been visited (even in previous visits), beyond simply saving the result. This is one reason why also the result of CNF transformation are memoized. Both the change of space complexity due to maximal sharing, and the use of memoization were already described by Goto (1974), where memoization is offered through the function assoccomp at page 17.

We also memoize the result of combining two formulas by $\wedge$ or $\vee$, but it seems that after introducing a global canonicalization hash-table, this might no more be

needed, if building the node takes only constant time – the saving might not be worth the effort; on the other hand, if rebuilding the node takes linear time, and this depends on the algorithm for set merging, then also this cache might be useful. However, it is a fundamentally different cache because it can at most save time linear in the number of children of a node.

A further problem is that even if results of CNF transformation are memoized, they must be propagated upwards multiple time; if the original formula tree is of exponential size, we still risk an exponential blowup.

Additionally, we need to be able to pretty-print formulas for the output. In a simple string representation, we concatenate the results of visiting subtrees, so the total length of the resulting string can be exponential. Indeed, when this was our actual implementation, on complex input files our tool often exhausted the available heap exactly in this phase. This problem was partially alleviated by simply avoiding this concatenation: we now print the string piece by piece, during the visit, and have introduced a special token type to represent formulas to avoid having a textual representation. However, this is the only token type which does not have a string representation, and this complicates the implementation of our preprocessor.

Furthermore, this only shifted the problem: while memory consumption remains reasonable, on the same inputs now TypeChef is happy to produce various gigabytes of output – even if these are not the typical cases.

We have two possibilities to avoid all these problems, both in CNF and stringification: the first is using the same idea as in formula renaming instead of replacing a reference to a macro with its macro body. Indeed, we do not even need to introduce

a new name: we can simply reuse the existing name of the referenced macro, to increase readability. This might reduce the possibilities of producing an exponential formula in the first place.

Indeed, expanding a formula into another is similar to inlining a function into another, as done by optimizing compilers, which face the same problem of avoiding exponential blowup when doing so; similar heuristics can be used. In our case, we found that simply expanding inline small formulas (smaller than a fixed threshold) is beneficial, because it allows further simplification.

When outputting a string, formula renaming needs to be slightly modified. CPP does not check satisfiability of a formula, it checks if it is satisfied. Therefore, to introduce a new variable, we need to output a #define statement.

The other possibility is to identify sharing explicitly and introduce new names. While it is not a problem for CNF transformation, for the output this would reduce readability, therefore we want to avoid it or leave it optional, and use it as a last resort. However, this is possible. We can modify the algorithm for a standard DAG visit, by keeping two sets: the sets of visited nodes, and the sets of nodes which where encountered in the visit when already present in the first set. The formulas in the second set could then be renamed, making the size of the output formula linear.

4.4.3 An exponential example

When the program includes a file conditionally multiple times, we have a concrete case which can trigger formula blowup. Below, we show a simplified example, which shows how formulas can grow; the resulting pattern would be simplified by

our simplification rules, but for more complex examples, too complex to present here, this is no more true.

```
1  #if A
2  //After expanding #include <header.h>
3  #ifndef HEADER_H
4  #define HEADER_H
5  //Header body
6  #endif
7  #endif
8  // Now defined(HEADER_H) = defined(A) || definedEx(HEADER_H) = φ
9  #if B
10 //After expanding #include <header.h>
11 #ifndef HEADER_H
12 #define HEADER_H
13 //Header body
14 #endif
15 #endif
16 // Now defined(HEADER_H) = φ ∨ (B ∧ ¬φ) =
17 // defined(A) || definedEx(HEADER_H) || (!defined(A) && !definedEx(HEADER_H) &&
       defined(B))
```

5 Conclusion and future works

5.1 Parsing

We have discussed the design and the challenges behind the design of our partial preprocessor. In this chapter, we briefly discuss how the output of partial preprocessing can be parsed and typechecked.

Baxter and Mehlich (2001) and Padioleau (2009) modify a standard C grammar to allow explicitly for CPP annotations at specific places; they note that with this approach, allowing preprocessor annotations at any source location would be unwieldy. The first problem is that preprocessor annotation can occur at arbitrary places; the second is that matching #if and #endif annotation might occur while processing different nonterminals. Thus, in such approaches, the grammar is modified to handle common patterns of preprocessor usage. However, Liebig et al. (2011) discuss such patterns of *disciplined annotations* and show that on average 16% of all annotations are not disciplined.

In a nutshell, instead of modifying the grammar, our approach is to modify the semantics of the grammar to encode once and for all the semantics of conditional

compilation: when the parser reads an #if directive, it should split the parsing context, producing two parsers. In the context of each parser we maintain the presence condition of the text which is being parsed. The two obtained parsers should then continue parsing the two possible alternatives; when an #endif directive is found, the two parsers should try to complete parsing a syntactic unit, reach a common position and then join together again. The resulting AST would contain a node representing the two alternatives, and the conditions leading to one or the other.

However, obtaining a robust algorithm to ensure that parsers join if possible is complex, and we do not detail this further. A further complication is that #if directives do not occur in isolation, but are often nested, so often split parsers need to split again; additionally, in practice parsers do not always join right after a #endif directive, therefore parser might split again also on non-nested conditions.

Therefore, a parser with associated presence condition γ, when reading an #if directive with condition φ, needs to check whether $\gamma \rightarrow \varphi$ or $\gamma \rightarrow \neg\varphi$ is valid; in this case, the parser needs only to follow the considered branch, otherwise it needs to split again.

5.1.1 Token stream representation

In our current representation, as described in our paper (Kästner, Giarrusso, and Ostermann, 2011), the output of the partial preprocessor is internally represented as a token stream where each token is associated with its presence condition in the output; thus, #if and #endif directives are represented only implicitly. However, in this implicit representation, it is no more immediate to understand where code

regions start and end. Consider now a generic directive of form #if φ in a region with p.c. ψ: in this representation, we simply annotate the contained tokens with p.c. $\varphi \wedge \psi$, while the context has p.c. ψ, and it is in general difficult to extract the *difference* of two formulas, obtaining again φ. Therefore, a parser with p.c. γ reaching such a directive will have to check validity of $\gamma \to (\varphi \wedge \psi)$ and $\gamma \to \neg(\varphi \wedge \psi)$, rather than $\gamma \to \varphi$ and $\gamma \to \neg\varphi$. Since ψ accounts for all outer #if directives, while φ is likely small, this makes a big difference; the expensive operation, during benchmark, was conversion to CNF, not validity checking (which invokes the efficient SAT-solver), and that conversion has already been performed for γ. Therefore, we believe that changing the representation might be a useful optimization, and we plan to do this in future work.

5.1.2 Typechecking

Having such a representation available, performing typechecking is a comparatively easier task. Variability-aware extension of existing type systems have been designed for instance by Kästner, Apel, Thüm, and Saake (2011), which introduced a general method which can be extended to other type systems as well.

5.2 Related work

Our work is most closely related to the one by Latendresse (2003, 2004): the algorithm he describes is similar to what we discussed in Sec. 3.2.

An interesting difference is that the representation he uses for formulas, which

he terms *conditional values*, most closely resembles an hybrid between BDD and boolean formulas. He does not detail which satisfiability procedure he used.

He argues on some examples that his algorithm has in practice linear time complexity; however, his example only show that unrelated macros do not interact and therefore the complexity does not grow with the number of macro bindings.

Additionally, his implementation is not tuned for performance and his evaluation is performed only on two very small examples. His algorithm always fully expands macro conditions, even if in a different representation, therefore we believe that making his algorithm perform well would require a substantial redesign, as needed for us.

Moreover, his algorithm only decides the presence condition of each line of source code, but does not perform partial preprocessing; macro expansion is limited to what is needed to process conditional compilation directives.

For further discussion of other related work, we refer to Kästner et al. (2011).

5.3 Future works

Some of the algorithms described for formula manipulation have not yet been fully implemented, and finishing this work is the obvious next step.

We already outlined the complete TypeChef project, which is nearing completion of its next milestone.

One interesting additional application of PPC would be to combine our work with the algorithms by Spinellis (2003): his algorithm tracks macro expansion in

a fully correct way to enable rename refactoring, but cannot cope with conditional compilation. On the other hand, PPC handles conditional compilation but needs to perform full macro expansion, thus combining the two tools would extend rename refactoring to software product lines.

Another possibility is extending the study of Chap. 3 to a full semantics and correctness proof of partial preprocessing.

Additionally, a last area for future work is independent caching of different headers. To speed up our tool, we could cache the result of partial preprocessing a header, because given the same macro table, the same preprocessor output and the same resulting macro table are produced; in particular, not the whole original macro table is relevant, and we can extract a partial macro table containing relevant entries. In C compilers, that is done only for the first header file, because as soon as any new macros are defined, the result might be different.

It could even be possible to partially preprocess each header in isolation, and then assemble and further preprocess the result. It is more perspicuous to regard this as partially evaluating each header in isolation (together with the recursively included files), and then performing further partial evaluation on the result. An essential condition for this to work is that each headers includes all the needed dependencies, rather than expecting some macros to be already defined by the including file; however, this is a well-known and widely respected guideline in many software projects, and exceptions might be indicated by the user.

Bigger technical problems are caused by the technicalities discussed in 3.1.1; in particular, preprocessing is not idempotent, as we discussed, so further preprocess-

ing a file is not necessarily safe. It seems possible that such partial preprocessing might depend on deferring some non-deferrable headers; thus, we might need the ability to defer more macro definitions. Both problems might be solved by using a more powerful output format; for instance, expanded self-references are marked as expanded during (partial) preprocessing to prevent recursion, and we need to preserve such marking also in the output. Details are yet to investigate.

5.4 Conclusion

This thesis discussed our partial preprocessing algorithm, why efficient formula manipulation is crucial, and how we can perform it. The analysis performed identified algorithms which cannot be made efficient, design requirements which had to be changed, and guided in designing better algorithms; the study that we performed about the connection with partial evaluation opens interesting possibilities for future works.

Acknowledgements

I would like to thank my supervisors during this thesis, Prof. Klaus Ostermann, Prof. Giuseppe Pappalardo, and Dr. Christian Kästner, for their suggestions and guidance during the work on this thesis and more in general.

I would also like to thank my colleagues Tillmann Rendel and Sebastian Erdweg for the many helpful discussions we had.

My work on this project was supported by the European Research Council (grant ERC #203099).

Personal acknowledgements follow in Italian only.

Ringraziamenti

A conclusione di questo lavoro, devo ringraziare coloro i quali mi hanno aiutato a compierlo, sia direttamente, sia indirettamente, aiutandomi a crescere e arricchendo quello che sono.

Ringrazio la mia famiglia, in particolare la mia cara e tenera madre Rosalia, mio padre Salvatore, mio zio Enzo, che mi hanno aiutato a crescere e diventare quello che sono.

Ringrazio il mio amico Matteo, per l'ineffabile amicizia che ci lega e per tutto il tempo passato assieme.

Ringrazio i miei amici Dario, Mauro, Silvia, Roberto, Bean, che mi hanno insegnato a non essere troppo serio e a sapermi sempre prendere in giro, e con cui ho la fortuna di poter parlare sempre schiettamente, senza segreti né ipocrisie.

Ringrazio i miei amici Enrico, Federica, Daniela, Pie(t)ro Torre, Simone, Domenico, Serena, Lorena, Toti, con cui abbiamo passato assieme tanti momenti belli della mia vita in questi anni.

Ringrazio tutta la SSC, in particolare Ivan, Salvo 'Spider', Roberta, Ivano, Beniamino, Mario, Alekampo, e tanti altri, per aver reso così belli gli anni vissuti nella

SSC.

Ringrazio i miei colleghi, Pino, Angelo, Eugenio, Samuele, Daniela, Lorenzo, Mikkule, Carlo, Gianluca, Giancarlo, Simone, Salvo ‘LtWorf’, per aver reso un po’ meno noiose tante giornate di studio.

Bibliography

Bram Adams, Wolfgang De Meuter, Herman Tromp, and Ahmed E. Hassan. Can we refactor conditional compilation into aspects? In *Proc. Int'l Conf. Aspect-Oriented Software Development (AOSD)*, pages 243–254, New York, 2009. ACM Press. ISBN 978-1-60558-442-3. doi: http://doi.acm.org/10.1145/1509239.1509274.

Sven Apel and Christian Kästner. An overview of feature-oriented software development. *Journal of Object Technology (JOT)*, 8(5):49–84, 2009.

A.W. Appel and M.J.R. Gonçalves. Hash-consing Garbage Collection. Technical Report CS-TR-412-93, Princeton University, 1993.

Ira Baxter and Michael Mehlich. Preprocessor conditional removal by simple partial evaluation. In *Proc. Working Conf. Reverse Engineering (WCRE)*, pages 281–290, Washington, DC, 2001. IEEE Computer Society. ISBN 0-7695-1303-4.

Claus Brabrand and Michael I. Schwartzbach. Growing languages with metamorphic syntax macros. In *Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 31–40, New York, 2002. ACM Press.

Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Computers*, 35:677–691, 1986. ISSN 0018-9340. doi: 10.1109/TC.1986.1676819. URL http://portal.acm.org/citation.cfm?id=6432.6433.

A. P. Ershov. On programming of arithmetic operations. *Commun. ACM*, 1:3–6, 1958. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/368892.368907. URL http://doi.acm.org/10.1145/368892.368907.

Jean-Marie Favre. Understanding-in-the-large. In *Proc. Int'l Workshop on Program Comprehension*, page 29, Los Alamitos, CA, 1997. IEEE Computer Society. doi: http://doi.ieeecomputersociety.org/10.1109/WPC.1997.601260.

Jean-Christophe Filliâtre and Sylvain Conchon. Type-safe modular hash-consing. In *Proceedings of the 2006 workshop on ML*, ML '06, pages 12–19, New York, 2006. ACM Press. ISBN 1-59593-483-9. doi: http://doi.acm.org/10.1145/1159876.1159880. URL http://doi.acm.org/10.1145/1159876.1159880.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Boston, MA, 1995. ISBN 0-201-63361-2.

Alejandra Garrido and Ralph Johnson. Refactoring C with conditional compilation. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*, page

323, Los Alamitos, CA, 2003. IEEE Computer Society. doi: http://doi.ieeecomputersociety.org/10.1109/ASE.2003.1240330.

Alejandra Garrido and Ralph Johnson. Analyzing multiple configurations of a C program. In *Proc. Int'l Conf. Software Maintenance (ICSM)*, pages 379–388, Washington, DC, 2005. IEEE Computer Society. ISBN 0-7695-2368-4. doi: http://dx.doi.org/10.1109/ICSM.2005.23.

Eiichi Goto. Monocopy and associative algorithms in an extended Lisp. Technical Report TR74-03, University of Tokio, 1974.

Ying Hu, Ettore Merlo, Michel Dagenais, and Bruno Laguë. C/C++ conditional compilation analysis using symbolic execution. In *Proc. Int'l Conf. Software Maintenance (ICSM)*, pages 196–206, Los Alamitos, CA, 2000. IEEE Computer Society.

ISO. *ISO/IEC 9899-1999: Programming Languages—C*. International Organization for Standardization, 1999.

Neil D. Jones. An introduction to partial evaluation. *ACM Computing Surveys (CSUR)*, 28:480–503, 1996. ISSN 0360-0300.

Christian Kästner, Paolo G. Giarrusso, Tillmann Rendel, and Klaus Ostermann. Variability-aware parsing in the presence of lexical macros and conditional compilation. In preparation.

Christian Kästner, Sven Apel, and Martin Kuhlemann. Granularity in software

product lines. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 311–320, New York, 2008. ACM Press. ISBN 978-1-60558-079-1. doi: http://doi.acm.org/10.1145/1368088.1368131.

Christian Kästner, Sven Apel, Salvador Trujillo, Martin Kuhlemann, and Don Batory. Guaranteeing syntactic correctness for all product line variants: A language-independent approach. In *Proc. Int'l Conf. Objects, Models, Components, Patterns (TOOLS EUROPE)*, volume 33 of *Lecture Notes in Business Information Processing*, pages 175–194, Berlin/Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-02570-9. doi: 10.1007/978-3-642-02571-6.

Christian Kästner, Sven Apel, Thomas Thüm, and Gunter Saake. Type checking annotation-based software product lines. *ACM Trans. Softw. Eng. Methodol. (TOSEM)*, 2011. accepted for publication.

Christian Kästner, Paolo G. Giarrusso, and Klaus Ostermann. Partial preprocessing of C code for variability analysis. In *Proc. Int'l Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, 2011. accepted for publication, preprint: http://www.informatik.uni-marburg.de/~kaestner/vamos11.pdf.

Maren Krone and Gregor Snelting. On the inference of configuration structures from source code. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 49–57, Los Alamitos, CA, 1994. IEEE Computer Society. ISBN 0-8186-5855-X.

Mario Latendresse. Fast symbolic evaluation of C/C++ preprocessing using condi-

tional values. In *Proc. European Conf. on Software Maintenance and Reengineering (CSMR)*, pages 170–179, Los Alamitos, CA, 2003. IEEE Computer Society. ISBN 0-7695-1902-4.

Mario Latendresse. Rewrite systems for symbolic evaluation of C-like preprocessing. In *Proc. European Conf. on Software Maintenance and Reengineering (CSMR)*, pages 165–173, Washington, DC, 2004. IEEE Computer Society. ISBN 0-7695-2107-X.

Jörg Liebig, Christian Kästner, and Sven Apel. Analyzing the discipline of preprocessor annotations in 30 million lines of C code. In *Proc. Int'l Conf. Aspect-Oriented Software Development (AOSD)*, 2011. accepted for publication.

Bill McCloskey and Eric Brewer. ASTEC: A new approach to refactoring C. In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*, pages 21–30, New York, 2005. ACM Press. ISBN 1-59593-014-0. doi: http://doi.acm.org/10.1145/1081706.1081712.

Marcílio Mendonça, Andrzej Wąsowski, Krzysztof Czarnecki, and Donald D. Cowan. Efficient compilation techniques for large scale feature models. In *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE)*, pages 13–22, New York, 2008. ACM Press.

Marcílio Mendonça, Andrzej Wąsowski, and Krzysztof Czarnecki. SAT-based analysis of feature models is easy. In *Proc. Int'l Software Product Line Conference (SPLC)*, pages 231–240, Pittsburgh, PA, 2009. Carnegie Mellon University.

A. Moors, F. Piessens, and M. Odersky. Parser combinators in Scala. CW Report CW491, Department of Computer Science, KU Leuven, 2008.

Andreas Nonnengart and Christoph Weidenbach. *Computing small clause normal forms*, volume 1, chapter 6, pages 335–367. Elsevier, Amsterdam, the Netherlands, 2001. Handbook of Automated Reasoning.

Yoann Padioleau. Parsing C/C++ code without pre-processing. In *Proc. Int'l Conf. Compiler Construction (CC)*, pages 109–125, Berlin/Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-00721-7. doi: http://dx.doi.org/10.1007/978-3-642-00722-4_9.

T. Troy Pearse and Paul W. Oman. Experiences developing and maintaining software in a multi-platform environment. In *Proc. Int'l Conf. Software Maintenance (ICSM)*, pages 270–277, Los Alamitos, CA, 1997. IEEE Computer Society. doi: http://doi.ieeecomputersociety.org/10.1109/ICSM.1997.624254.

Henry Spencer and Geoff Collyer. #ifdef considered harmful or portability experience with C news. In *Proc. USENIX Conf.*, pages 185–198, Berkeley, CA, 1992. USENIX Association. ISBN 1-880446-44-8.

Diomidis Spinellis. Global analysis and transformations in preprocessed languages. *IEEE Trans. Softw. Eng. (TSE)*, pages 1019–1030, 2003. ISSN 0098-5589.