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

Paolo G. Giarrusso

04 March 2011

Software product lines (SPLs)

SPL = 1 software project $\xrightarrow{\text{Feature selection}}$ 1 variant of a program, out of many possible ones.

Examples of features:

- Which data representation to use?
- Support end-user feature so-and-so?
- Fast or real-time version?

(Static) correctness checking

Aim: to support developers, check if all variants are “correct”

- Syntactic correctness
- Type-correctness
- Bug finding
- Static analysis
- Model checking (freedom from deadlock, liveness)
- ...

Exponential number of variants

33 optional, independent features $\Rightarrow$

a unique variant for each person on the planet

Slide credits: Christian Kästner

Exponential number of variants

320 optional, independent features $\Rightarrow$

variants $>$ # estimated atoms in the universe

Slide credits: Christian Kästner

Example SPLs

NASA flight control system: 275 features

Vim (text editor): 779 features

HP Owen printer firmware: 2000 features

Linux kernel: > 6500 features

Approach

Analyse the whole SPL at once!

Parsing: build a conditional AST, which stores the presence conditions (boolean formulas) of code elements

SPL-aware type checking: if `A` refers to `B`, `B` must be present whenever `A` is: $pc_A \rightarrow pc_B$.

If conflicting definitions are present, they must not be active at the same time: pc_A xor pc_B.

Done for other languages (e.g., Java)

Rely on SAT-solvers

We need therefore to check formula validity.

NP-complete problem! Exponential time again!

For many classes of problems, available SAT-solvers are efficient.

Our problem is one of those!

Conditional compilations for SPLs

Use a lexical preprocessor (like the C preprocessor, CPP) to implement SPLs.

Example:

```
1 #if FEATURE_REAL_TIME
2 void sort(int array[], int length) {
3   //Use heap sort, always O(n log n)
4 }
5 #else
6 void sort(int array[], int length) {
7   //Use quick sort, usually but not always faster.
8 }
9 #endif
```

Conditional compilation is available in other languages as well.

Analysis of unpreprocessed code

C compilers first preprocess code, then parse it.

Instead, we need to parse C code before preprocessing.

But it is hard!

CPP mixes variability with other stuff.

Examples for parsing CPP

```
#define P(msg) \
    printf(msg);

main() {
  P("Hello\n")
  P("World\n")
}
```

Macro expansion required for parsing!

```
#ifdef BIGINT
#define SIZE 64
#else
#define SIZE 32
#endif

allocate(SIZE);
```

Alternative definitions

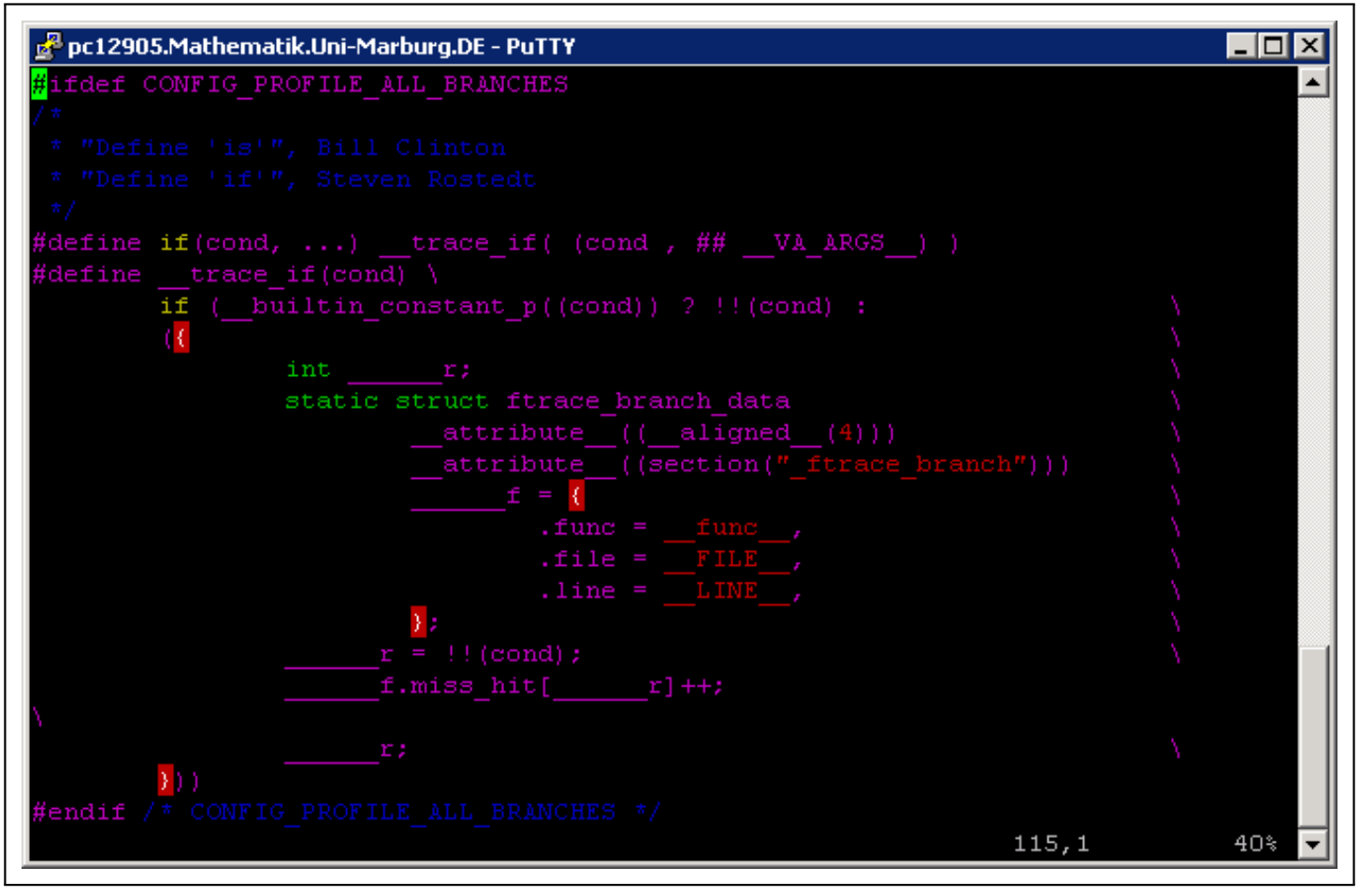
```
if (!initialized)
#ifdef DYNAMIC
  if (enabled)
#endif
    init();
```

Undisciplined annotations

(Around 16% in a study of 40 Open Source projects)

Slide credits: Christian Kästner

From the Linux kernel:



Slide credits: Christian Kästner

Requirements

The output must:

Be simple to further process (esp. parse)

Contain only variability, remove unrelated constructs

Avoid `#define` ...

⇒ use only `#if ...#endif`
⇒ ~~Avoid `#define`~~

⇒ Use only `#if...#endif` and `#define`

Correctness of partial preprocessing

Ideally, our correctness requirement would be:

$$cpp(\sigma, ppc(prog)) = cpp(\sigma, prog)$$

The actual specification is more complex and has quite a few restrictions, which are OK for our application scenario.

Conditional compilation

```
1 #if C_1
2 body 1
3 #elif C_2
4 body 2
5 #else
6 body else
7 #endif
```

becomes:

```
1 #if C_1
2 body 1
3 #endif
4 #if !C_1 && C_2
5 body 2
6 #endif
7 #if !C_1 && !C_2
8 body else
9 #endif
```

Macro expansion

Given:

```
1 #if C_1
2 #define A (expansion_1)
3 #elif C_2
4 #define A (expansion_2)
5 #endif
```

a reference to `A` becomes:

```
1 #if C_1
2 (expansion_1)
3 #endif
4 #if !C_1 && C_2
5 (expansion_2)
6 #endif
7 #if !C_1 && !C_2
8 A
9 #endif
```

Include guards

Typical header structure, for `foo.h`:

```
1 #ifndef FOO_H
2 #define FOO_H
3 /* Header body */
4 #endif
```

This way, multiple or even (indirect) recursive inclusions of `foo.h` are tolerated.

Therefore, when `FOO_H` is tested, we need to check if it is satisfiable $\Rightarrow$ again, use SAT!

Real-world example:

```
static void rt_mutex_init_task(struct task_struct *p)
{
        raw_spin_lock_init(&p->pi_lock);
#ifdef CONFIG_RT_MUTEXES
        plist_head_init_raw(&p->pi_waiters, &p->pi_lock);
        p->pi_blocked_on = NULL;
#endif
}
```

```
static void rt_mutex_init_task(struct task_struct *p)
{
#ifdef CONFIG_DEBUG_SPINLOCK
do { static struct lock_class_key __key; __raw_spin_lock_init((&p->pi_lock), "&p
->pi_lock", &__key); } while (0)
#endif
#ifndef CONFIG_DEBUG_SPINLOCK
do { *(&p->pi_lock) = (raw_spinlock_t) { .raw_lock =
#ifdef CONFIG_SMP
{ 0 }
#endif
#ifndef CONFIG_SMP
{ }
#endif
,
#ifdef CONFIG_DEBUG_LOCK_ALLOC
.dep_map = { .name = "&p->pi_lock" }
#endif
 }; } while (0)
#endif
;
#ifdef CONFIG_RT_MUTEXES
        plist_head_init_raw(&p->pi_waiters, &p->pi_lock);
        p->pi_blocked_on = ((void *)0);
#endif
}
```

Slide credits: Christian Kästner

The need for simplification

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

Define `B` as:

```
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
```

Simplified result

```
1 #if FEAT2 && FEAT1
2 BODY1
3 #endif
4 #if FEAT2 && !FEAT1
5 BODY2
6 #endif
```

Less duplicated literals (or none)!

Even more important in complex, real-world examples!

Scalability requirements

Potentially huge codebases (Linux kernel)

File inclusion: a file can include thousands of lines of extra code.

During development, naive algorithm implementation lead to:

- Filling up the disk (>9G of output for one file)
- Filling up the heap (2-3G of RAM)
- $\Rightarrow$ Non-termination

Most of this happened during formula manipulation.

All state-of-the-art algorithms (including the alternative to SAT-solvers, i.e. BDD) have exponential worst-case complexity.

Formula representation – I

1st idea: Represent formula by an unordered node-labeled tree, similar to AST; nodes represent `And`, `Or` and `Not` operations on the nodes.

2nd idea: Hash-consing: each formula is represented exactly once; after a formula is built, it is looked up in a canonicalization map to find an existing copy, which is used if available.

$\Rightarrow$ Formula comparison becomes $O(1)$.

$\Rightarrow$ Formulas are represented by DAGs, not trees, because subtrees can be shared.

Formula representation – II

Simplification during construction: simplification rules remove some redundant terms.

`And` and `Or` nodes contain sets of nodes. This removes duplicates and speeds up membership testing, which becomes $O(1)$.

Negation normal form (NNF): negation is pushed down to literals, using DeMorgan laws. This is done during formula construction: quite tricky to make it non-exponential.

- Simplification rules require $O(1)$ negation.

Some simplification rules

$$e \wedge \mathit{False} \mapsto \mathit{False}$$

$$e \wedge e \mapsto e$$

$$\ldots$$

$$e \wedge (e \wedge e') \mapsto e \wedge e'$$

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

$$e \wedge (e \vee o) \mapsto e$$

$$e \wedge (\neg e \vee o) \mapsto e \wedge o$$

Remove duplicates (see e) (at least “nearby” ones)!

The dual of each rule is also present.

Exponential replication

Below, size of T_i can be polynomial or exponential in i, depending on how size is measured:

$$T_1 = a \tag{1}$$
$$T_{i+1} = T_i \wedge (b \vee (T_i \wedge \neg c)) \tag{2}$$

The difference is in the expansion! T_i appears twice in T_{i+1}; T_1 appears (indirectly) 2^n times in T_{n+1}.

Represented as a tree, the number of nodes is exponential in n; represented as a DAG, the number of node is linear in n.

Express such a formula in the output without `#define` $\Rightarrow$ fully expand references.

We need to preserve sharing!

Formula renaming

Formula renaming: If φ appears twice in ψ, replace it by a variable A, and impose $A \leftrightarrow \varphi$.

This avoids replication, and produces an equisatisfiable formula.

$A \leftrightarrow \varphi$ can be further optimized to reduce the output size, we omit here the details.

This technique is also crucial for non-exponential transformation of formulas into CNF for SAT-solving.

Conclusion

We discussed variability analysis for C; within this context, our focus was on efficient algorithms for boolean formula manipulation.

Thanks to these algorithms, we believe we will be able to partially preprocess the whole Linux kernel and Vim, while considering the whole feature model.

These techniques might also be useful for source manipulation tools for C (e.g., refactorings).

Part of this work was published as:

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)*, pages 137–140, New York, 2011. ACM Press.