A VARIABILITY-AWARE FEATURE-REGION ANALYZER IN LLVM

FLORIAN SATTLER

Master Thesis

Chair of Software Engineering
Department of Informatics and Mathematics
University of Passau

Supervisor: Prof. Dr.-Ing. Sven Apel

2nd corrector: Prof. Christian Lengauer, Ph.D.

March 20, 2017

I was born not knowing and have had only a little time to change that here and there.

— Richard Feynman

Dedicated to my family.

ABSTRACT

In the analysis of software, researchers are often interested in specific code regions, for example, a region that corresponds to a feature of a program. However, identifying these regions is challenging and increases implementation effort. Analysis developers have to find all source-code regions corresponding to a certain region criterion, extract the important parts, and process regions before they can analyze them.

To support developers in these tasks, we propose and integrate the concept of an "interest region" into the LLVM compiler infrastructure. To this end, we implemented a framework, called VaRA, that separates the detection of desired regions from the analysis process via an abstract interface. The separation enables users to write their own analyses and to use any region detection that is offered by our framework. In addition, we enable the reuse of existing analyses when creating a new region-detection technique.

In this work, we describe the concept of "interest regions" in detail and describe how we implemented the concept in the LLVM compiler infrastructure. Furthermore, we demonstrate the applicability of our approach by implementing a region detection that extracts software features, meaning code that depends on configuration options, and by writing an analysis that detects control-flow interactions among regions. We then combine the region detection and the interaction analysis to detect interactions between different features.

We live in a society exquisitely dependent on science and technology,
in which hardly anyone knows anything about science and technology.

— Carl Sagan

ACKNOWLEDGMENTS

I would like to thank my family for supporting and believing in me during my master degree, and my friends for pushing me to give everything. A very special thanks goes to my girlfriend, who encouraged me when things were not working during my thesis. Furthermore, I would like to thank the whole Software Engineering group for instructive discussions and help when I needed them. Especially, I would like to thank Andreas Simbürger who always had a sympathetic ear for me and helped me with my questions regarding LLVM.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ACRONYMS

AST	abstract syntax tree
BB	basic block
CFG	control-flow graph
IR	intermediate representation
SSA	static single assignment form
CG	code generation
VaRA	variability-aware region analyzer

1 INTRODUCTION

Nowadays, programs often have millions of users and all of them have different expectations about how a program should support or even solve its tasks. Some users would have designed the user interface differently, others require new functionality to perform a certain task. To suit the needs of all users, developers add variability to their programs, allowing users to configure the program to their needs. This variability is often added by introducing a configuration knob that controls a certain feature, meaning a code region that implements a particular functionality of the software. For example, the office suite LibreOffice did a redesign of their graphical user interface, but because some users might not like the new one, they added functionality to stay at the old design[1]. A user can use the old interface or activate a knob in the settings to switch to the new design.

However, during software evolution, more and more of these configuration knobs get added to make the program customizable, despite that the growing complexity makes testing harder and in the end programs more error-prone [12]. This complexity arises from possible interactions among code regions, which belong to knobs, that cause misbehaviour [5]. For example, code that manages the old user design can interact with another code region that handles the displaying of different fonts, which causes the font to be drawn incorrectly. However, testing all different interactions that could occur among knobs is impractical, because only 10 different knobs would require more than 2^{10} independent tests. Since many knobs may not interact, actually testing all interactions independently is not required [10, 11]. The problem is that developers do not have sufficient tools to assist them by determining which configurations need to be tested and which ones do not. Furthermore, there are other problems, such as the impact of new code changes, where developers do not know which parts of the code are affected by a change. Developing such tools and analyses is complex and gets even more difficult when researchers want to support different programming languages.

1.1 GOALS

Our goal is to provide a framework within the LLVM compiler infrastructure that allows researchers to build language-independent analyses, such as an interaction analysis or a change-impact analysis [2]. We aim to reduce the overhead researchers have during the

1 LibreOffice redesign: `https://heise.de/-3613125`, source: Heise (2017-02-07)

analysis-development process, such as identifying interesting regions in source code or implementing utility data structures. That is why our framework provides debugging utilities, data structures, and offers abstract interface boundaries to separate parts of the analysis, allowing each part to be reused by other analyses.

Furthermore, since we use LLVM as a basis, developed analyses can easier be integrated into the compilation process or into LLVM libraries, which can be used to build tools.

1.2 CONTRIBUTIONS

The contributions of this thesis are an extension to the LLVM-based C/C++ compiler `clang` that allows researchers to extract code regions that represent feature implementations. Furthermore, we contribute the implementation of the basic structure of our framework, which include analysis/region abstractions, utility data structures, and graph visualizers for debugging. In addition, we also implemented a technique to detect code regions that relate to software features, and a language-independent interaction analysis.

This thesis forms the foundation of our analysis framework, which we plan to enhance for future software-engineering research.

1.3 OVERVIEW

We divided this thesis into four main parts. First, we introduce background knowledge to familiarize the reader with feature-oriented software, the concepts of data-flow analysis, and internals of LLVM. Second, in our main part about feature extraction, we describe how we modified `clang` to extract features from source code, present the structure and the concept of our framework, and explain how we implemented our feature-region detection within LLVM. Third, we evaluate our framework by implementing a language-independent interaction analysis with our framework. We use this analysis to detect interactions among features and, thus, demonstrate that the separation, between regions and analyses, of our framework works. Last, we summarize our work and present ideas how we can further enhance our framework.

2 BACKGROUND

In this chapter, we introduce three areas in more detail to give additional background information: We begin by explaining what a feature is from the feature-oriented software point of view and explain feature-related problems and terms. Then, we give an introduction into data-flow analysis and explain the schema behind it. At the end of this chapter, we give an introduction into the LLVM compiler framework, where we explain the later used intermediate representation in more detail.

2.1 FEATURE-ORIENTED SOFTWARE

Often, a program is seen as a monolith, which solves a single task, but, if we look closer, it is actually a set of different functionalities that work together to perform a series of varying tasks. We call these different functionalities, which are located in different parts of a program, some spread over the whole code base, *features*.

In the following section, we introduce the concept of a feature in more detail. Furthermore, we discuss the problem of feature interactions and describes software product lines.

2.1.1 *Features*

During software development, developers often introduce variability because they want to enable the user to customize behaviour. Developers offer the user ways to, for example, select different algorithms, choose different implementations, or an option to enable logging. All these different functionalities, implementations or configuration options can be combined under the term *feature*. To make it more precise we define a feature as follows, defined by Apel et al. [3]:

Definition 1. A *feature* is a characteristic or end-user-visible behaviour of a software system. Features are used in product-line engineering to specify and communicate commonalities and differences of the products between stakeholders, and to guide structure, reuse, and variation across all phases of the software life cycle.

The motivations why developers introduce features differ, some just add new functionality to the software, others want there software to be more customizable. These user-controlled features, represent knobs that can be can be switched on and off, are also referred to as configuration options.

Listing 1: Example of a feature code region that is controled by a macro.

```
1  #define COLOR
2
3  ...
4  #ifdef COLOR
5  ... color code ...
6  #endif
```

In general, there are different implementation mechanisms to add configuration options to a program; two examples are compile-time variables and load-time parameters. In the following sections, we shall introduce compile-time and load-time configuration options in more detail. We describe how features are located in source code and how features can interact with other features, potentially creating unexpected program behaviour. Furthermore, we discuss how too many features can make software unreliable and impact usability.

2.1.1.1 *Compile-time configuration options*

Compile-time configuration options give an user the possibility to configure a software during compile time. Build systems, such as `make` or `ant`, offer options to use different files for compilation, allowing for a very coarse-grained way to change the behaviour of a program. Another very common way in programming languages of the C-family is the preprocessor, that allows conditional compilation. That is, it fades in parts of the source code based on certain variables. Listing 1 shows a macro (`#define`) that is used to implement variability. The feature code in Line 5 is only present if the variable `COLOR` is defined, meaning the feature "COLOR" is enabled. The user configures the program by defining the needed configuration variables in a special header file, creating his own program variant. A program *variant* is one valid program configuration out of the configuration space, the set of all, possible program configurations.

2.1.1.2 *Load-time configuration options*

Another kind of configuration options are load-time configurations options.

These options change the execution behaviour of the program by providing environment variables or program arguments. A user executes, for example, the `Linux` command `ls`, but he wants to use the long listing format: so, he passes the *-l* argument to the program. At the beginning of the programs, there is dedicated code to parse these arguments and to set configuration variables that change the execution behaviour of the program by changing the control flow. In our example, `ls` prints the desired output, but formats it to a structured list when passing *-l* on the command line. Load-time configuration options allow the user to utilize one program in different ways, adjusting it to different situations without having to recompile it.

2.1.1.3 *Presence conditions*

Introduced configuration options change configuration variables and these influence control flow. For load-time configuration options this is usually implemented with `if` blocks, that can activate feature code if the condition holds. In our terminology, the condition of an `if` that decides whether the feature code gets executed, because its condition includes, at least, one configuration variable, is called *presence condition*. Figure 1 shows two examples of `if` blocks. The conditions of

```
if (COLOR) {
  // color feature code
}
```

```
if (COLOR && (LIST || READABLE)) {
  // mixed feature code
}
```

Figure 1: Example of two presence conditions.

the `if` statements are the presence conditions of the corresponding blocks. In the left Listing, there is a simple `if` block that gets activated in the case that the `COLOR` variable is set. The right example contains a more complex presence condition for which `COLOR` must be set and, at least, one of `LIST` or `READABLE`. These presence conditions determine the presence of the feature code, meaning they control whether the color-feature's code is executed.

2.1.1.4 *Feature Interactions*

Different features implement different functionality and sometimes they need to interact with each other. Sometimes, features still influence each other, although they do not explicitly interact. For example, to the *-l* option, `ls` also offers *-h* to print size parameters in a human-readable style. If both options are specified, the code that structures the line output takes different input from code that formats the size. So, the features interact to create the user-expected behaviour. Such interactions are not visible when we consider only one feature at a time. Furthermore, such interactions are not always known or intended by the programmer. For example, regions in the source code that implement features can influence control flow or change the current state of the program in an unintended way, which then can lead to unexpected [12] or erroneous behavior in other parts of the code or impact program performance [14, 15]. Apel et al. [3] define feature interactions as follows:

Definition 2. A *feature interaction* between two or more features is an emergent behaviour that cannot be easily deduced from the behaviors associated with the individual features involved.

An *inadvertent feature interaction* occurs when a feature influences the behaviour of another feature in an unexpected way (for example, regarding the expected control flow, program or data state, or visible behaviour).

As follows from the definition, interactions are not limited to two features. But even when all possible feature combination that are composed out of two features work, such as (COLOR + LIST, COLOR + READABLE, LIST + READABLE), combining three features can result in a program crash. Interactions among an unspecified number of features are called *n-way interactions* or higher-order interactions, where *n* is the number of features involved. Apel et al. [3] define *n-way interactions* as follows:

Definition 3. If n features interact, but none of their strict subsets, this is called a *n-way interaction*.

Feature interactions, in particular, the unknown and unexpected ones, can cause errors in programs and are hard to detect. We shall explain this problem in more detail in the next section.

2.1.1.5 *Problems introduced by feature interactions*

Very often software developers do not know which features interact. If these unknown interactions cause defects, debugging becomes very hard, because the developer often cannot infer what causes the bug or which other features have an impact. Furthermore, software developers often add new functionality and make their software adaptable, which increases the amount of features, making the problem even worse. In case the developer wants to check all pairs of features to locate his bug, he has to evaluate $f(f-1)/2$ different program runs, where f is the number of features. Real world software projects tend to use many features [13], for example, the Linux kernel with 6320 features [4]. And testing all possible configurations, which is infeasible for larger projects, would only checks for 2-way interactions. To find higher-order interactions, we would have to evaluate all n-way interactions, where each *n* could lead to $\binom{f}{n}$ interactions [3]. The number of features and feature interactions lead to an combinatorial explosion that cannot be managed by developers. Furthermore, the combinatorial explosion also impacts the analysis of programs, making classical analysis techniques impractical. Tools such as *iGen* [11] use randomness to mitigate this problem, finding a small set of configurations that contain "many" interactions but have no guarantee to find all. Another approach is use variability-aware analyses that exploit similarities among different variants [10].

However, not just the interactions but also the large number of features itself make it harder for users to configure their software [6] and can lead to errors due to invalid configuration. Xu et al. show in their work that a lot of the configuration parameters are not needed and their amount can be reduced without impacting users too much [16].

Handling a large number of features is difficult for developers and for users, because of the combinatorial explosion of the configuration space. That is why tool support is needed that can not only detect interactions between features but also make them manageable.

2.2 DATA-FLOW ANALYSIS

Control-flow graphs are a graph representation of a program. We describe how we use them to write a data-flow analysis. Furthermore, we introduce the terms interprocedural and context-sensitive in the context of data-flow analysis.

2.2.1 *Control-flow graph*

Plain source code is not a suitable program representation for our data-flow analysis, because we are interested in the relation between parts of the code and flow of information. These informations can be better represented in a graph that integrates them into its structure, a control-flow graph (CFG).

A CFG represent all possible paths within the program. Each graph node, called basic block (BB), is a block of instructions without any branches or jumps, except for the last instruction, the terminator. Only the terminator instruction can divert control flow from this BB to other BBs. Hence, the terminator defines the successors of a BB, either if it jumps to another BB or if another block directly follows [1]. Take,

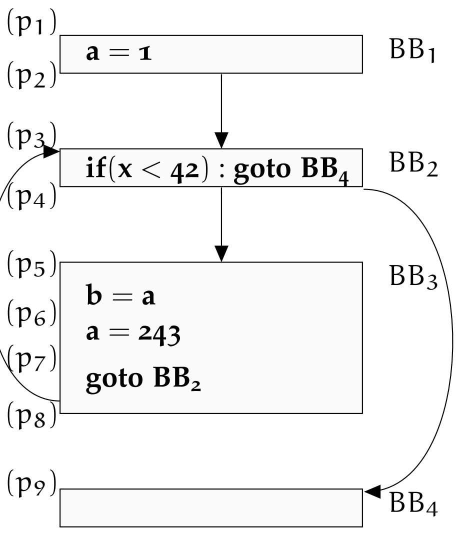


Figure 2: Control flow graph [1]

for example, the CFG for a simple program snippet in Figure 2. We begin the program by assigning 1 to the variable a, placing it within BB_1. Then we follow the control-flow edge from BB_1 to BB_2, where the program continues execution and checks if x is smaller than 42. At the end of BB_2 the control flow can either go to BB_3 or to BB_4, dependent on the terminator instruction. In case of BB_3 we get into a loop structure, where we execute the block and jump back to BB_2. In case of BB_4 we continue with the rest of the program.

Representing the program as a graph gives us the possibility to reason about control flow. How we can use this structure for data-flow analysis is described in the next section.

2.2.2 *Data-flow analysis*

Data-flow analysis is a technique to gather informations about a program, which integrates control-flow paths into the analysis to collect more precise information by relating it to specific locations in the program. The analysis inspects every instruction and aggregates information along the control flow. Hence, we need a fixed state between each instruction that represents the information.

Therefore, we define a program state as a finite number of facts (n) about the program. We place program states between each instruction and number them as $p_1, p_2, \dots, p_n$, $(1 \leqslant i \leqslant n)$. For example, we find 9 different states in our program in Figure 2. Each instruction then represents a transformation from one state to the next, meaning if p_i is the state before the instruction, p_{i+1} is the state after it. In order to set these states in relation, every data-flow analysis assumes there exists a data-flow value that is an abstraction of the set of all possible program states for that state. This allows us to define two set, for each instruction; IN, a set of data-flow values before the instruction, OUT, a set of data-flow values after the instruction. Furthermore, we create for each instruction two sets: gen, which represent the information generated by the instruction, and $kill$, which represents the deleted information. Then we define the transfer function as follows, taking IN as input and calculating OUT.

$$f(x) = gen \cup (x - kill)$$

This allows us to define the relation between IN and OUT sets within a BB. The OUT set of an instruction is the IN set of the following instruction. So, we can aggregate the information flow between instructions within a BB, by defining the IN and OUT sets of a BB. Its OUT set is the OUT set of the last instruction and its IN set is the IN set of the first instruction. After we have defined the information flow of a BB we now define how the sets of BBs relate. We calculate the IN set of a BB by combining all its predecessors (P) OUT sets with an analysis specific *meet* operator $\bigwedge$.

$$\mathrm{IN}[B] = \bigwedge_{p \in P} \mathrm{OUT}[p]$$

The well defined relations between instructions and BB allow us to propagate data-flow values through the program, which allows us to analyze it in more detail. However, we always have to consider that assumptions within our analysis can lead to incorrect results. So, we should interpret our results conservatively [1].

2.2.3 *Interprocedural and context-sensitive analysis*

Accuracy of the information gathered by an analysis depends heavily on properties such as scope and context. We can analyze every function separately but this makes our results imprecise, because by ignoring the relation between caller and callee we loose important information. The information difference arises from processing a call instruction by the transfer function. It can either incorporate the information from the OUT set of the other function or not. If the information is lost the data-values after the call are imprecise or just wrong. Hence, it can be necessary to analyze the whole program and preserve the connections between functions, creating an *interprocedural analysis*.

Another influence factor is the context in which a function is called, because the function behavior depends on the call site. If we want to eliminate this imprecision, introduced by not distinguishing between call sites, we have to analyze each function for every different call context, making our analysis *context-sensitive*.

However, making an analysis interprocedural and context-sensitive impacts the run time of the analyze and could make the processing of large software projects difficult. Hence, it is important to balance the precision and feasibility of an analysis.

2.3 THE LLVM FRAMEWORK

The following section introduces the design and concepts of the LLVM compiler framework. It describes the overall design and explains how the different components interoperate. Then we give an introduction to the intermediate representation (IR) used inside LLVM, called LLVM-IR. Afterwards, we describe the pass infrastructure and the C/C++ frontend clang.

2.3.1 *Design*

We begin our introduction to LLVM with an overall structure, shown in Figure 3. LLVM is split into three main components: frontend, optimizer, and backend.

First, the frontend reads a source file, parses it into an AST and performs language specific optimizations, such as type-alias analysis or constant folding. Afterwards, the frontend uses the AST to emit LLVM-IR, LLVM's intermediate representation that is used as common interface between the different components. As a second step, the optimizer takes the IR code from the frontend and performs language-independent optimizations. This is done by running different transformation passes over the IR code, such as constant propagation, loop-invariant code motion, and others. After all optimizer passes processed the IR code, it gets forwarded to the third component, the backend. The architecture-specific backend performs the target-specific optimizations and translates the IR code into machine instructions, for example, X86 or ARM.

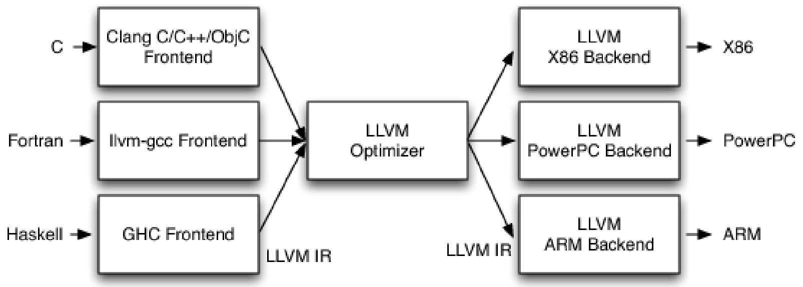


Figure 3: LLVM's three-phase design, showing a separation between frontend, optimizer, and backend [9].

As shown in Figure 3 the LLVM-IR decouples each component, by providing a common interface, allowing different front-/backends to be combined, and therefore, providing good modularization. This enables reuse not only for the front-/backends but also for the optimizer, meaning the structure of LLVM is language independent. For example, we can write a frontend for our own language that emits

LLVM-IR code and can immediately benefit from the implemented optimizations and different backends.

Language independence and a well designed IR together make LLVM an adaptable and thought-out compiler framework [8, 9].

2.3.2 *LLVM-IR*

LLVM-IR makes LLVM and all its passes language independent. In this section we give a deeper introduction into the IR, explaining basic concepts and the IR structure. We focus on how metadata is embedded and instructions that are particular important for our later implementation. In the end, we show how CFGs represent IR within LLVM.

2.3.2.1 *Introduction to LLVM-IR*

LLVM-IR serves as an internal representation of code for the compiler, allowing mid-level language-independent analysis and transformations but also aiming at being human readable to ease debugging.

The IR has a well-defined semantic, where instructions are represented in three-address form, meaning they have up to two operands and write there result in a different register. Take, for example, the two Listings 2 and 3, where Listing 2 represents the original source code in C++ and Listing 3 is the same in LLVM-IR. In our C++ example program, we create a variable, store 41 in it, add 1 to it, and return 0 from the main function. Listing 3 shows the same code in LLVM-IR, but is divided up in more instructions. Lets discuss the statically-typed IR in more detail:

Listing 2: C++ example

```
int main() {


  int var;

  var = 41;

  var += 1;

  return 0;
}
```

Listing 3: IR representation of the example.

```
1  define i32 @main() #0 {
2  entry:
3    %retval = alloca i32, align 4
4    %var = alloca i32, align 4
5    store i32 0, i32* %retval, align 4
6    store i32 41, i32* %var, align 4
7    %0 = load i32, i32* %var, align 4
8    %add = add nsw i32 %0, 1
9    store i32 %add, i32* %var, align 4
10   ret i32 0
11 }
```

In the first line, similar to C++, we define the main function, expressing the return type after the "define" with i32. LLVM-IR represents integers with arbitrary bit width, following the representation i{N}, where N is the number of bits in a range from 1 to $2^{23} - 1$. Line two is the label for the following BB "entry". At the start of the BB, Lines 3-4, two "alloca" instructions allocate memory on the stack for the variables `%retval` and `%var`. Then two "store" instructions write the initial values into the variables. For example, the "store" in Line 6 represents the assignment in the original program. The add instruction on Line 8 takes two operands, `%0` from the variable load in the line above, reading the current value from `%var`, and 1. Then, it adds both together and writes the result to a new temporary variable `%add`. We have to consider here that LLVM-IR is in static single assignment form (SSA), meaning that every value is defined before used and only assigned once. LLVM-IR implements this by using an unbounded set of temporaries that are numbered sequentially. Take, for example, the result of the "load" in Line 7, which is written to `%0`. To complete the += operation, the "store" instruction in Line 9 persists the result of the addition, which is stored in the temporary `%add`, in the variable `%var`. Finally, we complete the function by returning 0 from the main function in Line 10 [7, 9].

2.3.2.2 *Metadata*

Another important part of the LLVM-IR is its extensible metadata format, because it allows us to attach instructions with user-designed information. A metadata node is often used to encode debug informations, which can assist an analysis. Take, for example, Listing 4 with our "store" instruction from the previous example. A `!` indicates metadata, and the "FOO" identifier marks a named metadata node. The "!0" is used to reference the actual node containing the information, which can be found at the end of the file. The actual node is split from the instruction to allow reuse and save space. In our example the node itself is a string containing the word BAR.

Listing 4: Example of a metadata node in LLVM-IR.

```
  store i32 %add, i32* %var, align 4, !FOO !0
  ret i32 0
}

!0 = !{!"BAR"}
```

In order to create a metadata node and attach it to an instruction LLVM offers a well defined interface. Each metadata object is created by a factory, which tracks all nodes and tries to reuse them if a new one would contain the same information. If we want to create our BAR string from the example, we could use code as shown in Listing 5. We create an array `MA[]` of `Metadata *`, which contains an `MDString`. Then we create an `MDNode *` that holds the `MDString`, which then can be attached to an instruction with `setMetadata(StringRef Kind, MDNode *Node)`. The `StringRef Kind` represent the name of the metadata node `"FOO"`.

Listing 5: Creating a metadata node containing a string.

```
llvm::Metadata *MA[] {
  llvm::MDString::get(LLVMContext, "BAR");
};
Instruction->setMetadata("FOO", llvm::MDNode::get(LLVMContext, MA));
```

This composition scheme, composing higher-level metadata objects out of pointers to other metadata objects, allows us to build flexible, but at the same time lightweight, metadata representations, which we later use to add information about features.

2.3.2.3 *Important instructions*

In this section, we describe parameter attributes and the instructions, "alloca", "load", "store", "br", and "call" of LLVM-IR in more detail, because we use them later in our analysis. Listing 6 shows all instructions that we introduce in a small example program.

Listing 6: Program with example instructions.

```
%var = alloca i32, align 4
%array = alloca [5 x i32], align 16
store i32 5, i32* %var, align 4
%0 = load i32* %var, align 4
%1 = icmp ne i32 %0, 0
br i1 %1, label %then, label %else

%2 = load i32* %var, align 4
%call = call double @_Z3fooi(i32 %2)
```

PARAMETER ATTRIBUTES

LLVM-IR uses parameter attributes to attach additional information to function parameter types and return types. The following list describes a subset of attributes, that can be attached to some of the later described instructions.

`byval`: indicates that the attributed parameter, which is of type pointer, should actually be passed by value, for example, to pass an array or struct by value. This attribute is restricted to LLVM pointer arguments and we have to assume that it belongs to the caller not the callee.

`sret`: hints that the accessed pointer points to the address of a structure that is later returned from the function. This attribute cannot be placed at return types.

`align <n>`: guides the optimizer to assume that the given pointer value is `<n>` byte aligned.

INSTRUCTION: ALLOCA

```
<result> = alloca <type> [, <ty> <NumElements>] [,align <n>]
```

Allocation instruction, or short "alloca", reserves memory on the stack of the current function, which shall be automatically released at the end of the function. We set the memory size by providing a `<type>` and the number of elements, which is by default one, specifying that `sizeof(<type>)` $*$ `<NumElements>` bytes should be allocated. The `<result>` of the instruction is a pointer to the address of the allocated memory of type `<type>` $*$ [7]. For example, Line 1 in Listing 6, allocates space for a 32-bit integer with a 4-byte alignment, that means, the stack pointer shall be on a 4 byte aligned address. In Line 2 we allocate an array of 5 integers, so $5 *$ `sizeof(i32)` $= 20\ \mathit{bytes}$.

INSTRUCTION: LOAD

```
<result> = load [volatile] <ty>, <ty>* <pointer>[, align <alignment>]
```

Load instruction, or short "load", loads a value from the specified memory address and puts it in a new temporary variable(`<result>`). In cases were the load is marked `volatile` the compiler cannot change the order of execution of the volatile loads. As an example we can look at the "load" in Line 4, which loads the value at the address of the variable `%var` into the temporary `%0`.

INSTRUCTION: STORE

```
store [volatile] <ty> <value>, <ty>* <pointer>[, align <alignment>]
```

Store instruction, or short "store", writes a `<value>` to a memory location specified with a `<pointer>`. Important to remark here is that `<value>` and `<pointer>` need to have the same first class type (`<ty>`). The same compiler constrains that apply to volatile "loads" apply also to volatile "stores". In our example in Listing 6, we use the store in Line 3 to write 5 into the variable saved at the address `%var`.

INSTRUCTION: BR

```
br label <dest>
br i1 <cond>, label <iftrue>, label <iffalse>
```

Branch instruction, or short "br", diverts the control flow from one BB to another BB, marked with the correct label. Control flow shall be discussed in more detail in the next section. The branch can either be unconditional, always resulting in a jump to the `<dest>` block, or conditional. Every conditional branch checks whether `<cond>` is 1. In case it is the control flow is diverted to `<iftrue>`, otherwise to `<iffalse>`. The last three lines of our example in Listing 6 show how a branch works. First, in Line 4 we load a value from `%var`. Second, we make an integer compare between the loaded value and `zero`, writing the result to `%1`. Third, the branch instruction checks whether the compare returned `one` or `zero` and branches accordingly.

INSTRUCTION: CALL

```
<result> = call <ty>|<fnty> <fnptrval>(<function args>)
```

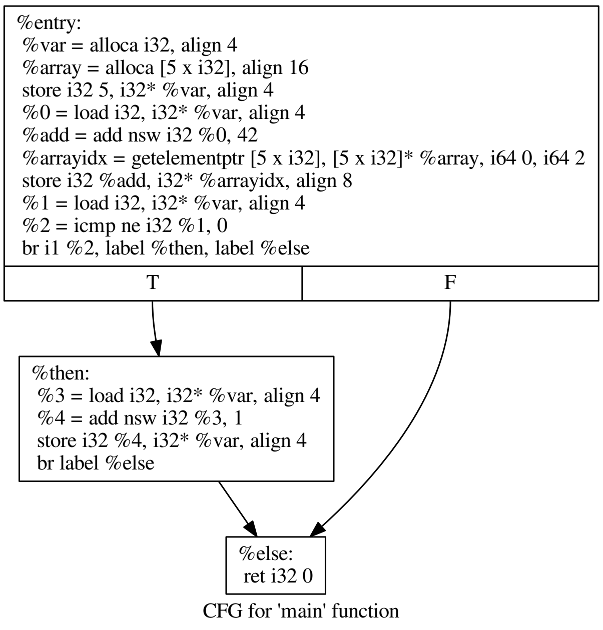
Call instruction, or short "call", represents a call to another function, meaning it diverts the control flow to the other function and marks the continuation point after the function returns. The first two parameters represent certain types of the function; `<ty>` represents the type of the call instruction and the return value of the function, where `<fnty>` is the signature of the function that is called, which is only required if the function has variable arguments. Next `<fnptrval>`, is a pointer to the function that shall be called. Then the parameters list `<function args>` follows, specifying the values that are passed to the function. Furthermore, the provided parameters have to match the signature of the function even if not specified. In our example 6 we call a function `@Z3_fooi` in line 9. As parameter we pass and 32-bit integer `%2`, the value loaded in the line above. The result of the function call will be of type double and stored in the local register `%call`.

2.3.2.4 *Control-flow in LLVM*

After we introduced a few special instructions, we now turn our focus to the overall structure of the LLVM-IR. On the highest level LLVM defines the term `module` to represent a translation unit. Translation unit is the whole input c/cpp file, after the preprocessor has been executed, which is then given to the compiler to be compiled into an object file. A module consists of symbol-table entries, global variables, and a list of functions. Each function is composed out of BBs, which are a list of instructions without a branch in-between, meaning that only the last instruction, the so called `terminator`, can divert control flow. The function defines an entry BB, where control flow starts when the function is called. Every BB has a set of successor blocks where the control flow can go; this is defined by the `terminator`. Take, for example, a branch instruction. It has two possible BBs it can jump to, the then or the else block, making them a possible successor of the BB. Another `terminator` instruction is "ret", the return instruction, marking the end of a control-flow within a function and returning to the callers control flow.

This structure of a function is at the same time the control-flow graph, because it represents the control flow through the function. To visualize the graph, we use the predefined graph printers which LLVM offers through the option `-dot-cfg`. We use the example in

Listing 6 from before to generate via the `opt`-tool the control-flow graph, shown in Figure 4. The first BB of our graph contains the same



```
%entry:
 %var = alloca i32, align 4
 %array = alloca [5 x i32], align 16
 store i32 5, i32* %var, align 4
 %0 = load i32, i32* %var, align 4
 %add = add nsw i32 %0, 42
 %arrayidx = getelementptr [5 x i32], [5 x i32]* %array, i64 0, i64 2
 store i32 %add, i32* %arrayidx, align 8
 %1 = load i32, i32* %var, align 4
 %2 = icmp ne i32 %1, 0
 br i1 %2, label %then, label %else
```

```
%then:
 %3 = load i32, i32* %var, align 4
 %4 = add nsw i32 %3, 1
 store i32 %4, i32* %var, align 4
 br label %else
```

```
%else:
 ret i32 0
```

Figure 4: Example control-flow graph.

instruction sequence as shown in Listing 6. In the end of this BB, we can take two control-flow edges, either to the `%then` block or `%else`. If the branch condition is true, the next block is `%then`, where `%var` is increased by 1, in case it is false, or after we exit the `%then` block, we branch to the `%else` block. When the control flow reaches the last instruction of the `%else` block, zero is returned and we switch the control flow back to the caller.

2.3.3 *Pass Infrastructure*

The LLVM optimizer is the middle component of the compiler, which aims to make the IR representation of the program more efficient. These optimizations are applied in different passes, where each pass does transformations or computes information. For example, the dead instruction elimination (`-die`), which removes instructions that are never executed, or the promote memory to register pass (`-mem2reg`), which changes memory references to register references by making memory accesses local. In general, there are two categories of passes: analysis passes, computing higher-order information about the program (for example, information about pointer aliasing) or transformation passes, which transform the IR code to make it more efficient. Each pass can specify other passes that should be run before it, because it depends on the calculated information or assumes that some transformations were run before it. This creates dependencies between passes and can make it necessary to run passes more than once, because some passes may invalidate the information computed by another pass. Each pass specifies if it "preserves" the computed information or if not, requiring the reexecution of the pass that calculated it. This is why passes are registered with a `Passmanager`, which manages the dependencies and schedules passes.

Let us take a closer look at how a pass works and how we can create our own pass within LLVM, so we can later write our own analysis. First, we have to define on which levels we want our pass to run, for example, each function or each BB. We specify this by inheriting from one of the different pass classes, for example, a `llvm::ModulePass` runs once on every module where a `llvm::LoopPass` processes every loop of a program independently. Second, we implement the basic interface by creating a static variable `ID` and overwriting the runOn* function, for a module pass this would be: `bool runOnModule(Module &M)override`. Last, we now register our new pass with the LLVM `Passmanager` by instantiating the register template: `static RegisterPass <OurPass> X("OurPass name", false, false)`. In addition, we could implement the `getAnalysisUsage` method to specify our dependencies to other passes. With only a few steps, we create our own pass and place it in the LLVM pass infrastructure, allowing us to write a powerful analysis without much engineering effort.

2.3.4 *Frontend clang*

After we explained the LLVM-IR and the LLVM pass infrastructure within the optimizer, we now turn to the language dependent part of the LLVM frontend. Regarding frontends, we introduce `clang`[1] in more detail, but other frontends work in similar ways. `Clang` is LLVM's frontend for parsing C based languages, supporting C, C++, Objective C/C++, and OpenCL C. The main task of the frontend is to translate the source code into LLVM-IR.

`Clang` processes the code in three steps. First, the preprocessor resolves `#include` directives and does expansions of `#define` and `#ifdef` macros. Second, the `clang`semantic engine, internally called `Sema`, creates the abstract syntax tree (AST). The language-dependent AST is composed out of many different nodes, which are grouped into three base classes: (1) declarations, such as variable declarations (`VarDecl`), (2) statements, such as a `BinaryOperator`, and (3) types. Take, as example, the C++ code in Listing 7, which corresponds to our IR example from before. We generate a visualization of the AST by calling `clang` with `-Xclang -ast-dump`. Figure 8 shows a section of the AST dump, omitting unimportant typedefs. On the top level is a `FunctionDecl` for

1 http://clang.llvm.org/index.html

Listing 7: C++ source code corresponding to our previous IR example.

```cpp
double foo(int a) {
  return 0.0;
}

int main() {
  int var = 5;
  int array[5];

  if (var)
    foo(var);

  return 0;
}
```

our main function. Two nodes deeper into the AST we find a `DeclStmt` for our variable declaration in Line 2 of the source file. Both sub nodes describe the declaration in more detail, the `VarDecl` describes the variable and the `IntegerLiteral` represents our initial value of 5. Another interesting node is the `IfStmt`, which represents our `if` in Line 6. After the two <<<NULL>>> lines, we find the condition (`ImplicitCastExpr`), the then block of the if (`UnaryOperator`) and the last <<<NULL>>> represents the non-existent else statement. Third, after the frontend created the AST, it traverses the AST to create corresponding LLVM-IR code. After code generation, `clang` passes the code to the optimizer.

Listing 8: Example AST produced by clang.

```
FunctionDecl 0x164d221 <example.cpp:1:1, line:10:1> line:1:5 main 'int (void)'
`-CompoundStmt 0x164d748 <col:12, line:10:1>
  |-DeclStmt 0x164d3c8 <line:2:3, col:14>
  | `-VarDecl 0x164d348 <col:3, col:13> col:7 used var 'int' cinit
  |   `-IntegerLiteral 0x164d3a8 <col:13> 'int' 5
  |-DeclStmt 0x164d4b8 <line:3:3, col:15>
  | `-VarDecl 0x164d458 <col:3, col:14> col:7 used array 'int [5]'
  |-BinaryOperator 0x164d610 <line:5:3, col:20> 'int' lvalue '='
  | |-ArraySubscriptExpr 0x164d560 <col:3, col:10> 'int' lvalue
  | | |-ImplicitCastExpr 0x164d548 <col:3> 'int *' <ArrayToPointerDecay>
  | | | `-DeclRefExpr 0x164d4d0 <col:3> 'int [5]' lvalue Var 0x164d458 'array'
        'int [5]'
  | | `-IntegerLiteral 0x164d4f8 <col:9> 'int' 2
  | `-BinaryOperator 0x164d5e8 <col:14, col:20> 'int' '+'
  |   |-ImplicitCastExpr 0x164d5d0 <col:14> 'int' <LValueToRValue>
  |   | `-DeclRefExpr 0x164d588 <col:14> 'int' lvalue Var 0x164d348 'var' 'int
       '
  |   `-IntegerLiteral 0x164d5b0 <col:20> 'int' 42
  |-IfStmt 0x164d6d8 <line:6:3, line:7:8>
  | |-<<<NULL>>>
  | |-<<<NULL>>>
  | |-ImplicitCastExpr 0x164d678 <line:6:7> '_Bool' <IntegralToBoolean>
  | | `-ImplicitCastExpr 0x164d660 <col:7> 'int' <LValueToRValue>
  | |   `-DeclRefExpr 0x164d638 <col:7> 'int' lvalue Var 0x164d348 'var' 'int'
  | |-UnaryOperator 0x164d6b8 <line:7:5, col:8> 'int' postfix '++'
  | | `-DeclRefExpr 0x164d690 <col:5> 'int' lvalue Var 0x164d348 'var' 'int'
  | `-<<<NULL>>>
  `-ReturnStmt 0x164d730 <line:9:3, col:10>
    `-IntegerLiteral 0x164d710 <col:10> 'int' 0
```

3 DETECTING FEATURES IN LLVM-IR

In this chapter, we explain how we use our framework to extract features from source code and make them available in LLVM. First, we describe how we modified the C/C++ frontend `clang` to extract features from source code and annotate them in LLVM-IR during compilation. Second, we explain our framework called VaRA (variability-aware region analyzer) in more detail and highlight what makes it adaptive to create new analyses and existing analyses reusable. Third, we present our feature detection as an example for a detection analysis that can be written with VaRA.

3.1 FEATURE EXTRACTION WITH CLANG

Before we can analyze features, we first have to locate them in the source code and mark their locations in LLVM-IR. In Section 2.1.1, we introduced features in software and explained that load-time options are represented by `if` conditions. In this section, we locate these `if` blocks in the AST of a program and transfer them into a data structure, representing presence conditions. We then describe our metadata format and demonstrate how we create it out of the presence condition. Furthermore, we explain how we attach metadata to LLVM-IR BBs to track features. We implement our feature extraction for C/C++ programs by modifying `clang`, but the general idea can be implemented in all LLVM frontends.

3.1.1 *Locating features in C/C++ ASTs*

Previous to the extraction, `clang` has to load a configuration file that contains the names of the configuration variables. Currently, this configuration file is just a list of names; however, we implemented an interface for accessing the configuration that allows us to replace the simple file format with a more detailed one. During the initialization of the `CodeGenModule` which handles the generation of code for a module, `clang` initializes our `CodeGenFD`. The `CodeGenFD`, which is short for code generation feature detection, automatically loads the configuration file and is meant to track the found feature occurrences.

To locate features, we hook into `clang`s code generation, especially the function `EmitIfStmt` within the `CodeGenFunction` class which handles the code generation (CG) of a function. Listing 9 shows a code snipped we want to process, which corresponds to the AST in Listing 10. `EmitIfStmt` is called to generate the IR code for an `IfStmt` AST node, Listing 9: Code example with two features.

```
if (FOO) {
  if (BAR) {
    // feature code
    a += 42;
  }
}
```

for example, the two in Listing 10 (Line 1, 7). Therefore, it is a good

Listing 10: Clang produced AST for Listing 9

```
1  IfStmt 0x2140c98 <line:8:3, line:13:3>
2   |-<<<NULL>>>
3   |-<<<NULL>>>
4   |-ImplicitCastExpr 0x2140b48 <line:8:7> '_Bool' <LValueToRValue>
5   | `-DeclRefExpr 0x2140b20 <col:7> '_Bool' lvalue Var 0x2140650 'FOO' '_Bool'
6   |-CompoundStmt 0x2140c78 <col:12, line:13:3>
7   | `-IfStmt 0x2140c40 <line:9:5, line:12:5>
8   |   |-<<<NULL>>>
9   |   |-<<<NULL>>>
10  |   |-ImplicitCastExpr 0x2140b88 <line:9:9> '_Bool' <LValueToRValue>
11  |   | `-DeclRefExpr 0x2140b60 <col:9> '_Bool' lvalue Var 0x2140718 'BAR' '
       _Bool'
12  |   |-CompoundStmt 0x2140c20 <col:14, line:12:5>
13  |   | `-CompoundAssignOperator 0x2140be8 <line:11:7, col:12> 'int' lvalue '+=
       ' ComputeLHSTy='int' ComputeResultTy='int'
14  |   |   |-DeclRefExpr 0x2140ba0 <col:7> 'int' lvalue Var 0x2140a88 'a' 'int'
15  |   |   `-IntegerLiteral 0x2140bc8 <col:12> 'int' 42
16  |   `-<<<NULL>>>
17  `-<<<NULL>>>
```

extension point to determine if the `IfStmt` is related to feature code. First, we call the function `CodeGenFD::getHeadFeatureMD`, forwarding the `IfStmt`. The function tries to create a `PresenceCondition`, our abstraction to represent presence conditions, for the `IfStmt`, which is only successful if the statement contains at least one configuration variable. We traverse all variables in the condition of the `IfStmt` and check if a variable name matches with one we previously loaded from the configuration file. In the example, we find a `DeclRefExpr` within the first `IfStmt`, where the used variable name (`FOO`) matches with a configuration variable. Thus, we create a `PresenceCondition` based on the `IfStmt`. Furthermore, the found `if` block needs to be annotated with feature information, hence we push the `PresenceCondition` onto a stack to keep track of it. Keeping a stack is important because, if we have two nested `ifs`, the presence condition of the inner one must also satisfy the outer condition, meaning that we have to combine them with a logical AND (&&). Listing 9 shows two nested `if` blocks. Here, the presence condition of the outer `if` is `FOO`, where the presence condition of the inner one is `FOO && BAR`, different from its actual condition `BAR`.

In the other case, we find no matching variable and therefore do not create a `PresenceCondition`. We just return a `nullptr`, which gets ignored from the rest of the code, and are done with processing this `IfStmt`.

Before we continue to explain how we annotate feature related BBs we describe our metadata format in the next section.

3.1.2 *Feature metadata*

We want to annotate each BB related to a feature with metadata that provides information about the feature block, which are either expensive to be recalculated or would get lost during the transformation to LLVM-IR. `CodeGenFD` offers a function `getCurrFeatureMD`, which creates a `llvm::MDNode` from the `PresenceConditions` on the stack. The metadata node is based on the grammar represented in Listing 11, beginning with start symbol ⟨*FMDNode*⟩. Every ⟨*FMDNode*⟩ is composed out of

Listing 11: Metadata grammar

```
⟨ FMDNode ⟩   ::=  ⟨ FBlock ⟩  |  ⟨ FBlock ⟩  "&"  ⟨ FMDNode ⟩
⟨ FBlock ⟩    ::=  "["  ⟨ Type ⟩  ","  ⟨ PC ⟩  "]"
⟨ PC ⟩        ::=  ⟨ Var ⟩  |  ⟨ Var ⟩  ⟨ Op ⟩  ⟨ PC ⟩  |  "("  ⟨ PC ⟩  ")"
⟨ Var ⟩       ::=  "{"  ⟨ VarName ⟩  "}"  |  ⟨ Neg ⟩  "{"  ⟨ VarName ⟩  "}"
⟨ Type ⟩      ::=  "H"  |  "T"  |  "E"
⟨ Op ⟩        ::=  "&&"  |  "||"
⟨ Neg ⟩       ::=  "!"
⟨ VarName ⟩   ::=  $VARNAME
```

at least one ⟨*FBlock*⟩, which corresponds to one feature block. Each ⟨*FBlock*⟩ consists of a ⟨*Type*⟩ and the presence condition of the block (⟨*PC*⟩). The presence condition of each ⟨*FBlock*⟩ is a sequence of variable names that can be surrounded with brackets, negated, and are joined by different operands — either AND (&&) or OR (||). ⟨*Type*⟩ indicates the part of our feature block; `"H"`, meaning this is part of the header/condition, `"T"` representing the "then" part, and `"E"` representing the "else" part of the `if`. For example, we create the metadata string `[H,{FOO}]` for the BBs corresponding to the `if` condition in lines four and five of Listing 10. Regard to the nested `if` we produce two ⟨*FBlocks*⟩ and join them by an AND (&), resulting in `[T,{FOO}] & [H,{BAR}]`. For this case, we used only one & to make parsing the string easier, meaning one & joins different ⟨*FBlock*⟩ where two are used within presence conditions.

Currently, our metadata node is implemented to produce a `llvm::MDString`, but this can later be replaced by adding a special `MDNode` to LLVM. A special `llvm::FeatureMDNode` would preserve the semantics behind our grammar and reduce the complexity of parsing feature metadata.

3.1.3 *Annotating LLVM-IR with feature metadata*

We continue with the execution of `EmitIfStmt` after we created the `PresenceCondition`. Now, we need to mark each BB that is related to a feature with a metadata node, but BBs themself cannot be annotated with metadata. Thus, we use the terminator of a BB to attach our metadata. Because there are different terminator instructions and we have no direct access to the inserted instructions in the `EmitIfStmt` function, we separate the insertion of the metadata node from `EmitIfStmt`. At every location a terminator is created, we add code to insert our metadata. Listing 12 shows the code we added to the `EmitBranch` function. The first line of our change tries to create a metadata node by

Listing 12: Annotating branch instructions with metadata.

```
    ...
    auto BInst = Builder.CreateBr(Target);

+   llvm::MDNode *Feature = CGM.getCurrFeatureMD();
+   if (Feature)
+     BInst->setMetadata("Feature", Feature);
  }
```

calling `getCurrFeatureMD`. In case this is successful, we add the node to the instruction created earlier; otherwise we do nothing. Besides the branch instruction, we also have to add this code to other terminators such as return, unreachable or switch instructions.

However, separating the insertion of metadata requires us to track in which branch of the `if` we currently are. To achieve this, we add the state to the `PresenceConditions` on the stack. The state tracks which branch of the `IfStmt`, corresponding to the `PresenceCondition`, was already processed. During the execution of `EmitIfStmt`, we call three functions, provided by `CodeGenFD`, to update the state. First, `finishHead` after the code for the `if` condition was emitted. Second, `finishThen` after we processed the "then" block. Last, we call `closeFeature` to remove the `PresenceCondition` from the stack. This allows us to precisely add metadata information to every instruction we want and still keep track of the metadata we have to generate.

With all these adjustments we can automatically locate, create, and annotate LLVM-IR with feature metadata. Listing 13 shows part of the generated IR code from our example in Listing 9. We see at the end of the `entry` block, the feature metadata for the condition of `if (FOO)`. Another example is `!Feature !3` that marks the "then" block of the inner `if (BAR)`. It has two conditions `FOO` and `BAR`, resulting in the combined presence condition `FOO & BAR`.

Listing 13: LLVM-IR with feature metadata for the code in Listing 9.

```
entry:
  ...
  br i1 %tobool, label %if.then, label %if.end3, !Feature !1

if.then:
  ...
  br i1 %tobool1, label %if.then2, label %if.end, !Feature !2

if.then2:
  ...
  br label %if.end, !Feature !3

if.end:
  br label %if.end3, !Feature !4
  ...

!1 = !{!"[H,{FOO}]"}
!2 = !{!"[T,{FOO}] & [H,{BAR}]"}
!3 = !{!"[T,{FOO}] & [T,{BAR}]"}
!4 = !{!"[T,{FOO}]"}
```

3.2 VARIABILITY-AWARE REGION ANALYZER

This section introduces the idea and the structure of our variability-aware region analyzer (VaRA). We begin by defining the concept of a region within VaRA. Then, we explain the interface offered by VaRA in more detail and introduce our generic visualizer.

3.2.1 *Region concept behind VaRA*

The general idea behind VaRA is to extend LLVM with an interface that allows researchers to write arbitrary analyses, based on LLVM-IR. For example, we could write an analysis pass that tries to detect interactions between features, to combat the problem mentioned in Section 2.1.1.5. Furthermore, we aim to make our analyses reusable, so that we can use the same analysis on different sections of the code. Hence, we have to decouple the analysis from the type of region that is analysed. To this end, we define the concept of an *iregion*, which represents a section in the code we are interested in for analysis. For our interaction example, we would define an iregion as a code section belonging to a feature.

We implemented this concept in VaRA in the class `vara::IRegion`. `IRegion` provides an abstract interface for handling different kinds of regions, which shall be explained in the next section in more detail. The developer creates his own region class that represents a section relevant to him and inherits from `IRegion`. This provides him with a default implementation that eases debugging, such as graph printers, and speeds up development due to pre-implemented data structures. Furthermore, `IRegion` sets a clear interface for analyses already implemented in VaRA. This means that the developer can use these analyses on his regions, by only implementing the interface. In addition, this also helps analysis writers, because they can use different regions to run their analysis on. To complete our example, we implement a region class that is based on `IRegion` and we implement an analysis that detects interactions between `IRegions`. This enables us to find feature interactions, but also allows us to combine features with other analyses and find interactions between other types of regions.

3.2.2 *IRegion interface of VaRA*

After we explained the concept behind an `IRegion`, we now focus on the technical part. We describe some implementation details and highlight how they allow us to use VaRA. Listing 14 shows part of the `IRegion` interface. We first notice that `IRegion` is actually a `class template`, which can be specified by providing a type for `PART`. This is necessary because of two reasons; first, different types of regions can have different types of parts. Second, an `IRegion` manages the parts it consists

Listing 14: IRegion interface

```
class IRegionBase {
public:
  unsigned int getID();
};

template <typename PART>
class IRegion : protected IRegionBase {
public:
  std::string getName();
  unsigned int getID();

  IRegionKind getKind() const;

  virtual void dump(bool full = false);

  bool isTopLevel() const;
  void addSubRegion(IRegion *IR);
  llvm::SmallVector<IRegion *, 4> getSubRegions();

  SubIRegionIter SubRegion_begin();
  SubIRegionIter SubRegion_end();

  IRegionIter begin();
  IRegionIter end();

  bool contains(PART *BB) const;
  virtual bool isInPosPart(PART *BB) const;
  virtual bool isInNegPart(PART *BB) const;

template <typename PART>
class IRegionDetection : public FunctionPass {
public:
  virtual llvm::SetVector<IRegion<PART> *> getRegions() = 0;
```

of for the user and therefore needs the type of a part. For example, one user needs regions based on `llvm::BasicBlocks`, whereas another wants to group `llvm::Functions` together. Furthermore, `IRegion` also has a base class (`IRegionBase`) that, despite different template instantiations, provides an unique `ID` for every `IRegion`.

We now focus on the public methods of `IRegion`. There are different getters for utility informations, such as name or `ID`, but also a special getter `getKind`. `IRegionKind` is an enum that is used for LLVMs own runtime type information (RTTI), to allow for more efficient down casting. This works similar to calling the instance-of operator in Java before casting. We provide this interface to be compatible to LLVM and to allow the user to utilize the `llvm::dyn_cast<>` macro, which allows him to cast an `IRegion` to a sub type. Next, the `IRegion` interface offers methods to add sub regions and to iterate over them, which is useful to nest regions. Furthermore, there are methods such as `contains` to interact with the parts of a region and iterators to iterate over all parts.

However, before we can analyze `IRegions`, we first have to create them. To achieve this, VaRA offers the `IRegionDetection` base class that itself is based on LLVM passes like `llvm::FunctionPass` explained in Section 2.3.3. Every detection pass that inherits from `IRegionDetection` provides a `virtual getRegions` function to access its found regions.

The `IRegion` interface combined with own detection passes completes our structure to analyze generic types of regions. It makes the designer of analyses and regions independent of each other, allowing reuse of existing implementations. Currently, this interface was sufficient for our implementations, but we plan to enhance it further to offer more support to analysis developers.

3.2.3 *IRegion visualizer*

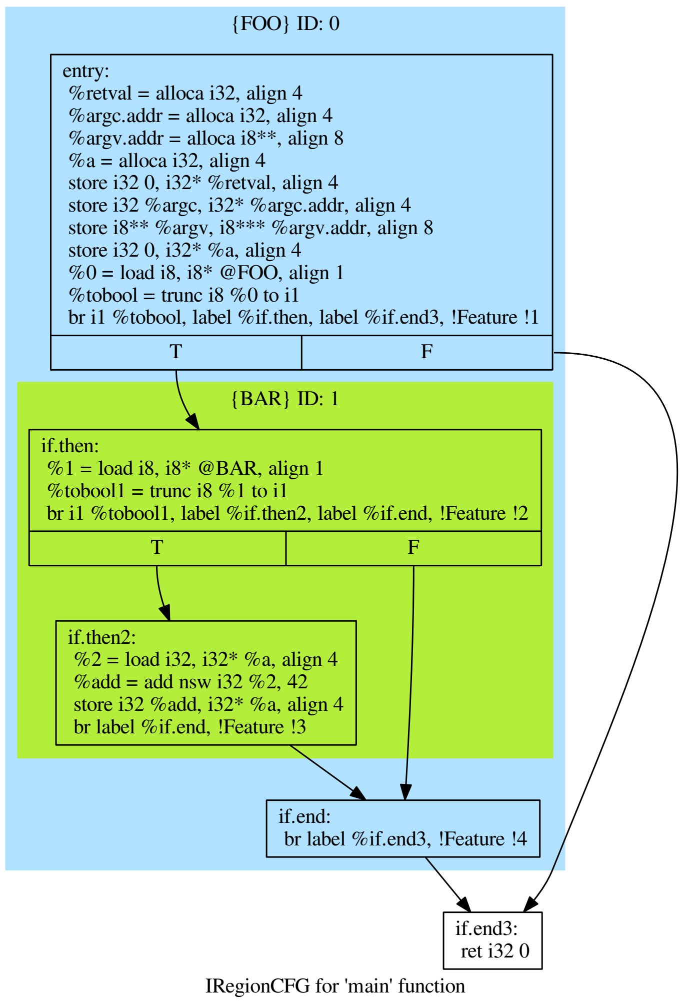
As an example, we describe `IRegionCFG`, a visualizer for CFGs that highlights regions. VaRA offers other support, such as special data structures and debugging utilities, too. The `IRegionCFG` eases debugging during the development of new region classes. It shows a CFG of a function and simply highlights the regions that are currently detected, so that the developer can easily see if all BBs are correctly grouped. To use the `IRegionCFG` the developer only needs to create a `IRegionFunction`, initialize it with the function he wants to draw, and add `IRegions` to it. Then he can call `viewCFG` to show the full CFG with details, or `viewCFGonly` to get a small version of the graph. Listing 15 shows how to draw an `IRegionCFG` and Figure 5 shows an example graph.

Listing 15: Code to view `IRegionCFG`

```
llvm::Function F;

IRegionFunction IRF = IRegionFunction(F);
IRF.addIRegion(/* pass IRegion*/);
IRF.viewCFG();
IRF.viewCFGonly();
```

In order to draw such a graph, VaRA uses `GraphTraits` provided from LLVM. `IRegionFunction` is a decorator around a `llvm::Function` that provides region information. In essence, it is a `struct` that holds the functions and pointers to all regions. We then specialize the two graph templates `GraphTraits<>` and `DOTGraphTraits<>` for our decorator `IRegionFunction`. `GraphTraits<>` specifies how our graph is connected and how we can iterate over the nodes. `IRegionCFG` is based on the CFG provided by LLVM. Therefore, we inherit the relations from `GraphTraits <llvm::BasicBlock *>` and provide our own entry node and node itera-



```
entry:
 %retval = alloca i32, align 4
 %argc.addr = alloca i32, align 4
 %argv.addr = alloca i8**, align 8
 %a = alloca i32, align 4
 store i32 0, i32* %retval, align 4
 store i32 %argc, i32* %argc.addr, align 4
 store i8** %argv, i8*** %argv.addr, align 8
 store i32 0, i32* %a, align 4
 %0 = load i8, i8* @FOO, align 1
 %tobool = trunc i8 %0 to i1
 br i1 %tobool, label %if.then, label %if.end3, !Feature !1

if.then:
 %1 = load i8, i8* @BAR, align 1
 %tobool1 = trunc i8 %1 to i1
 br i1 %tobool1, label %if.then2, label %if.end, !Feature !2

if.then2:
 %2 = load i32, i32* %a, align 4
 %add = add nsw i32 %2, 42
 store i32 %add, i32* %a, align 4
 br label %if.end, !Feature !3

if.end:
 br label %if.end3, !Feature !4

if.end3:
 ret i32 0
```

Figure 5: Visualization of the example code in Listing 9.

tor. We simply forward them to the iterators of the function stored inside the `IRegionFunction`, because we want the same relations between the BBs. The second template we have to specialize is `DOTGraphTraits` that specifies how the contents of a ".dot" file is created. To get a basic functionality, we inherit from `DOTGraphTraits<Function *>` and overwrite only methods where we want different contents. For example, we overwrite `getEdgeSourceLabel` to change which branch edges are drawn and `addCustomGraphFeatures`. This method is used to adapt the graph above-anticipated modifications by allowing write access to the `GraphWriter`, meaning we can directly write to the ".dot" output file. The ".dot" file specification allows us to cluster nodes into subgraphs and fill the cluster with a specified color. Hence, to color our regions, we create a string that describes a region as a subgraph and write it directly to the ".dot" file.

The developer can now automatically print a CFG that highlights its specified regions as a ".dot" file and thus visualize the internal structure, for example, by converting it to a "png" file[1].

1 dot -T png /PATH/iregioncfg:main.dot -o iregioncfg.png

3.3 FEATURE EXTRACTION IN LLVM

After the introduction of the generic structure of VaRA, we explain how we use it to create a feature extraction. This continues after clang generated LLVM-IR code, as describe in Section 3.1.3, and passed it to the optimizer. First, we describe the overall structure of features within VaRA and how they fit into the region concept. Second, we explain how we recreate features from the annotated metadata left by clang.

3.3.1 *The feature structure within VaRA*

We implemented the concept of a feature and their source code regions within VaRA. The following section explains the different classes used to make VaRA feature aware.

3.3.1.1 *Feature*

A feature is represented by two classes: one is the `FeatureVariable` that encapsulates the configuration variable. Every `FeatureVariable` has an unique identifier and a name, corresponding to the name of the configuration variable. Furthermore, every feature variable stores a list of `FeatureRegions` that belong to it, because they are influenced by the encapsulated configuration variable. The second class, `FeatureRegion` that is based on `vara::IRegion<llvm::BasicBlock>`, is a group of `llvm::BasicBlocks` that relate to the feature code. More precisely, the base class `IRegion` keeps track of all BBs in the regions, where the `FeatureRegion` itself groups the blocks into head/then/else to differentiate the parts of an `if` block. Hence, `FeatureRegion` implements the different methods to fulfill the `IRegion` interface, such as `contains` and `classof` (for RTTI), but also offers special methods to get more detailed information about a feature like if a block belongs to the condition part or a branch. Furthermore, each `FeatureRegion` has a `PresenceCondition` and a list of pointers to `PresenceConditions` from surrounding regions.

3.3.1.2 *Presence conditions*

`PresenceCondition` represents the condition under which the BBs in the "then" group of a `FeatureRegion` are executed. We structured our `PresenceCondition` class similar to the $\langle PC \rangle$ non-terminal in the grammar shown in Listing 11. Therefore, `PresenceCondition` is an abstract base class and can be either a `PresenceConditionValue` (`PCValue`) or a `PresenceConditionNode` (`PCNode`). A `PCValue` represents a single `FeatureVariable`, where a `PCNode` is a conjunction of two `PresenceConditions`. `PCNode` corresponds to a node in a binary tree, because it has a left- and a right-child node that are joined by an operator, which can be either AND or OR.

3.3.1.3 *FeatureManager*

`FeatureManager` is the core component of the feature support for VaRA, not only because it manages features and provides an interface to access features, but also because it serves as a mapping between LLVM data structures and VaRA. Because our feature classes have a lot of relationships among each other, we need a component to ensure that certain invariants, which we assume between our data structures, are preserved. For example, a `FeatureRegion` can have multiple presence conditions, due to surrounding regions, which themself can have multiple `FeatureVariables`, such that a `FeatureRegion` belongs to different `FeatureVariables` and each of these `FeatureVariables` needs a connection back. Furthermore, we need mappings from LLVM classes to VaRA classes, such as the mapping from `llvm::Functions` to `FeatureRegions`. Hence, we restrict the creation of `FeatureRegions` to the `FeatureManager`, which is done by making the constructor private and declaring the `FeatureManager` as a friend, and let him manage all `FeatureVariables`. This allows us to assume certain invariants: they hold for the empty set, `FeatureManager` can ensure that they hold after modification, and other modifications are excluded. In addition, we can use the `FeatureManager` to handle memory management.

Furthermore, the `FeatureManager` offers an interface. For example, `getFRsForFunction` returns all regions belonging to a function or `getVFFromString` returns all `FeatureVariables` within a `string`, which can be helpfull for parsing metadata.

3.3.2 *Recreating feature regions from metadata*

In Section 3.1.3, we used `clang` to annotate LLVM-IR with metadata. This IR code is then passed to the optimizer where we want to analyze feature regions. Thus, we first need a LLVM pass that extracts the metadata and creates `FeatureRegions`, providing them to other analysis passes.

3.3.2.1 *Feature detection*

The `FeatureDetection` pass is based on the `vara::IRegionDetection` pass, which is a function pass. Therefore, the `FeatureDetection` processes each function by itself. It initiates the `FeatureManager` and initializes a `PresenceConditionMap`, which acts as a cache so we do not have to recreate `PresenceConditions` for metadata string we have seen before. Then, the `FeatureDetection` iterates over all BBs, and checks whether the block has feature metadata. In case it has metadata and this BB was not processed before, that is, it is not in the `visitedBBs` set, the `FeatureDetection` starts to extract a `FeatureRegion`.

3.3.2.2 *Creating a presence condition*

Before we continue with the extraction, we first need to describe how we convert the metadata representation of a feature into a `PresenceCondition`. The method `cPCFS()`[2] strips the header information from the string, by removing the parts that do not belong to the presence condition, and tries to create a `PCNode` from the string. To create a `PCNode`, we call the function `createPresenceConditionNodeFromString` that searches for junctions, for example, (&&) or (| |); furthermore, it tries to find enclosing brackets. We compute substrings based on whether we found a conjunction or brackets and there locations. Then, we either recursively call `createPresenceConditionNodeFromString` twice, one with the substring for right-hand side(rhs) the other with the sub string for left-hand side(lhs), or we call `cPCVFS()`[3]. In the latter case, a `PresenceConditionValue` is created and returned; by checking for if we need to create brackets or negation (!) markers, extracting the name of the feature variable from the string, fetching the `FeatureVariable` from the `FeatureManager`, and assembling a new `PresenceCondition`. In the recursive case, we combine the return values of rhs and lhs into a new `PresenceConditionNode` and return it. At the end, we recursively put together a `PresenceCondition` that represents the presence condition in the metadata.

3.3.2.3 *Creating a feature region*

As input to the constructor of a `FeatureRegion`, we get the created `PresenceCondition`, the entry block of region, and a set of BBs that have already been visited (`visitedBBs`). To create the region we have to find all BBs that belong to it, therefore, we create a queue that tracks all BBs we need to check and initialize it with the entry block. Then, we process every BB in the queue, skipping those we have processed before, so that they are in the set of visited BBs. We start by adding the BB into the set of visited BBs and check if the metadata of the BB describes the same or a subblock of the current presence condition. In the case it does, we get the group type (head/then/else) from metadata and add the BB into the corresponding group. Next, we queue every successor block of the BB to the queue for processing and continue with the next BB. After the queue is empty, the creation of the `FeatureRegion` is finished.

3.3.2.4 *Extracting a new feature region*

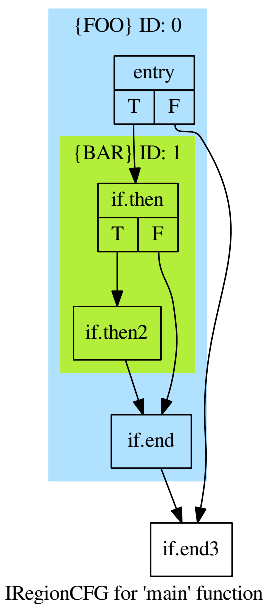
We begin the extraction of a `FeatureRegion` by fetching the metadata from the BB and converting it into a list of `PresenceConditions`. Corresponding to the nesting, the metadata string is split into different blocks; each of them is then converted into a `PresenceCondition`.

2 `createPresenceConditionFromString()`
3 `createPresenceConditionValueFromString()`

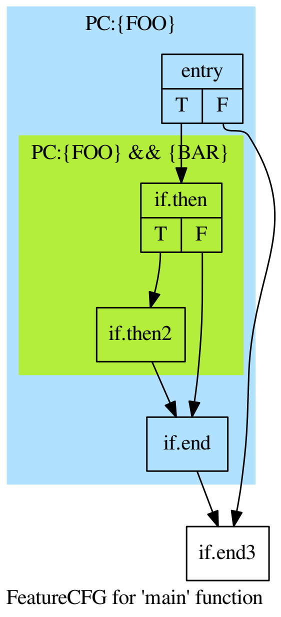
Since there are many equal blocks and we do not want to convert the same string twice, we use a cache to look up previously created `PresenceConditions`. Then we forward the list of `PresenceConditions`, the BB we are processing, and a list of visited BBs (`visitedBBs`) to the `createFeatureRegion` method of the `FeatureManager`. This function creates a new `FeatureRegion` and updates the data structure in the `FeatureManager`. First, a pointer to the new regions gets added to every region that surrounds it. Second, we add a mapping of from the `llvm::Function`, the BB belongs to, to `FeatureRegions` into a map. Third, the new `FeatureRegion` gets added to every `FeatureVariable` that occurs in its `PresenceCondition`. At the end we return a pointer to the new `FeatureRegion`.

3.3.2.5 *Visualizing feature regions*

After we have extracted all regions that correspond to features and created `FeatureRegions` for them, we now use VaRA's visualizer to show a simple CFG that highlights the found regions, as shown in Figure 6a.



(a) Visualization of the example code in Listing 9.



(b) `FeatureCFG` for the example code in Listing 9.

However, we can also adapt the code of the visualizer to create our own `FeatureCFG`, that contains a better representation for features. For example, we can add the full presence condition of the feature to the top of region, as shown in Figure 6b.

4 EVALUATION

In this chapter, we evaluate our framework VaRA by implementing a language-independent interaction analysis. We show how VaRA allows our analysis to only work on abstract regions by implementing the analysis only based on the `IRegion` interface. After that, we also evaluate whether our interaction analysis produces correct results by analysing and verifying certain example programs.

4.1 FEATURE-INTERACTION ANALYSIS

In the previous chapter we made the claim that VaRA allows developers to write analysis that run on LLVM-IR without considering on which kind of region they operate. To prove this, we demonstrate our feature-interaction analysis that we developed with VaRA. The analysis is only based on `IRegions` and finds interactions between them. For example, we shall use `FeatureRegions` to find interactions between different features. We begin with an explanation of the analysis-graph structure that is later used. Then we explain how we detect interactions based on `IRegions` but use it to find feature interactions.

4.1.1 *Analysis graph structure*

Our interaction analysis runs as a module pass in LLVM and requires two other passes as dependencies. `DominatorTreeWrapperPass` provides us with dominator information about BBs, meaning we can query if BB A dominates BB B. LLVM implements the relationship "dominates" as follows: BB A dominates B if and only if every control-flow path that leads to B must go through A. The other pass we depend on is `FeatureDetection` that provides us with region information, but here we could use any `IRegionPass`. For each function we only run the `FeatureDetection` and access it through the `IRegionPass` interface with `getRegions`, to get a list of all `IRegions`. This list is then forwarded to the top-level node of our analysis structure, which consists of `ModuleNodes`, `FunctionNodes`, and `BBNodes`.

4.1.1.1 *ModuleNode*

`ModuleNode` encapsulates a module and manages all relationships between functions. We add a function to the `ModuleNode` with the method `addFunction(Function *F, Set<IRegion> Regions)` that takes the function together with a set of `IRegions` as inputs. The `ModuleNode` then creates a

`FunctionNode` to represent the function and keeps a mapping between function and `FunctionNode`. In addition the `ModuleNode` also tracks all found interactions with an `InteractionStore`, allowing him to output all found interactions at the end or draw a graph to visualize the analysis structure. After we added all functions to the `ModuleNode` we call `analyze`. The method starts by pushing all functions onto a work queue and then starts to process the queue. For every interaction we pop the first function from the queue, lookup the corresponding `FunctionNode`, and call its analyze method. After the queue is empty the analysis of this module is finished.

4.1.1.2 *FunctionNode*

`FunctionNode` is a wrapper around a function that manages the relationships between the BBs. It creates a `BBNode` for every BB and holds a mapping between the nodes and BBs. Analog to the data-flow analysis schema from Section 2.2.2 every `FunctionNode` has an IN and a OUT set. These sets contain all `Accesses` that flow into or out of the function.

An `Access` is an abstraction of an memory operation that sums up information about the memory access like; which instruction accessed memory, in which region was the access, and did the operation read or write. Furthermore, it tracks a history of changes, meaning if the contents of location a was stored to b and b is then stored to c the history of the write `Access` to c includes this change, allowing us to track the data flow back to a.

Thus, IN is a set of `Access` object that represent an access to one of the variables that get passed into the function as a parameter and OUT is a set of `Accesses` that relate to the variable which is returned from the function. Similar to the `ModuleNode` the `FunctionNode` provides an `analyze` method that handles the analysis of the function. The work queue of a `FunctionNode` is initialized with every BB of the function and processes every BB by calling its analyze method. After the queue is empty the method checks if its OUT set has changed and in case it did all functions that depend on this one are added to the queue of the `ModuleNode`.

4.1.1.3 *BBNode*

`BBNode` is a small wrapper around a `llvm::BasicBlock` that also has an IN and a OUT set. However, it also has a third set of `Accesses` (Own-Accesses) that are created by the instructions of the BB. The relationships between BBs and there successors are handled as described in Section 2.2.2 with the union ($\cup$) as *meet* operator. How the analyses of a BB works is explained in detail in the next section.

4.1.2 *Detecting data-flow dependencies between features*

In this section, we describe the interaction detection. To find interactions we need to find data flow from one region to another, that means in general one region writes a value that later is used by another region or influences the regions control flow. In particular the three instructions "load", "store", and "call" are important to detect interactions, where the "call" instruction is only used to propagate access information between function calls.

The `analyze` method of the `BBNode` needs to compute and propagate variable accesses encapsulated by `Access`. This corresponds to the schema from the data-flow analysis Section 2.2.2; in our case we take the `Accesses` from the IN set, generate new `Accesses` from our "store" instructions (gen), delete accesses if we overwrite their information (kill), and forward the union of the IN set minus the deleted `Accesses` to the OUT set. The `analyse` method processes every instruction of the BB in the order they would be executed.

4.1.2.1 *Processing a "load" instruction*

In case the instruction is a "load" we check whether its operand `<pointer>` reads from a variable that is marked by an `Access`, either contained in the IN set or the set OwnAccesses. If it does and the BB belongs to a region we found an interaction, for which we create an `Interaction` object that we store in the `InteractionStore` of the `ModuleNode`. The `Interaction` class stores all important informations about the interaction, like which regions interact or the BB.

4.1.2.2 *Processing a "store" instruction*

If the instruction is a "store" and the BB is within a region we create a new `Access` to mark the write to the variable used in operand `<pointer>`. However, operand `<pointer>` might be a "load" from another location, therefore, we have to traverse the chain of "load" instructions to find the corresponding stack allocation, which we then use to mark the access. Next, we delete all `Accesses` from the IN set and the OwnAccesses set that are overwritten by the new `Access`. Then we need to check whether the "store" used an input that is marked by an `Access`, meaning operand `<value>` reads from a previously accessed value. In case we find an `Access` we create another separate new `Access` and preserve the information from the other access by adding it to the history of the new `Access`. This allows us to ensure the transitive access relationship, meaning if a is written within region FOO, then the value of a is read and stored in b, and after that b is used as an input for an instruction in region BAR, we get an interaction between FOO and BAR. Listing 16 shows an example of a transitive access relationship.

Listing 16: Transitive flow of information between feature regions.

```
  if (FOO) { // Feature Region FOO
    a = 41;
  }
  b = a + 1;
  if (BAR) { // Feature Region BAR
    if (b == 42) {
      // ...
    }
  }
```

4.1.2.3 *Processing a "call" instruction*

The handling of "call" instructions is split into two parts, handling the propagation to a function and forwarding the access returned from the function: first, we need to determine which `Accesses` get propagated to the function. We match the call parameters, the values passed to the function, to the function parameters, the values symbolizing the parameter within the function, and create new `Accesses` for the new variables, to preserve the relations to the old `Accesses`. Listing 17 shows an example where we pass the variable `callvar` to the function `foo` for the function parameter `funcvar`. This means we need

Listing 17: Difference of call and function parameter

```
void foo(int funcvar) { /* code */ }

{
  int callvar = 42;
  foo(callvar);
}
```

to create a new `Access` whose value is based on the allocation of the function parameter, fully preserving the history of previous accesses to the variable behind the call parameter. These new `Accesses` then get added to the IN set of the function we want to call. Furthermore, we register the function as dependent function with our `FunctionNode` and queue the function in the `ModuleNode` to be analyzed if the IN set of the other `FunctionNode` changed. Second, we need to forward the `Accesses` that are returned from the function into the current context of the `BBNode`. We iterate over the OUT set of the function and create new `Accesses`, preserving the history, in case the returned value is used.

4.1.2.4 *Completing the analysis*

After all instructions of the BB are processed we propagate all `Accesses` from the OUT set of this `BBNode` to its successors IN sets. In case this changed the IN set of the successor we queue for reanalyzing at the `FunctionNode`. The analysis finishes when each queue is empty and we do not have to reanalyze anything, meaning our analysis has reached a fix point regarding the information collected with `Accesses`. At the end of the analysis, we print all found `Interactions` that are stored in the `InteractionStore` of the `ModuleNode`.

To implement this feature-interaction analysis we did not have to use any `FeatureRegion` specific methods. Hence, our interaction analysis is independent of the type of region and we could use any region that is based on the `IRegion` interface.

4.1.3 *Current limitations*

In this section, we describe current limitations of our analysis. We discuss two problems that arises from the current implementation of our analysis structure. Furthermore, we explain the problem our analysis has when analysing pointers.

4.1.3.1 *Context-sensitivity*

The first problem is that the analysis currently is context insensitive, because we do not distinguish between different call locations and group them into one `FunctionNode`, making the results not wrong but imprecise. This can be solved by creating a `FunctionNode` for every call side of a function but this also impacts the speed of the analysis. Therefore, we need to develop a method to make the analysis context-sensitive without increasing the run-time too much.

4.1.3.2 *Inter module dependencies*

Another problem is that our analysis can only operate on a module level, meaning we cannot analyze function calls that call functions from another module, again making our analysis imprecise. This problem can partially be solved by reanalyzing these functions during link-time, but this works only if the function comes from another module of the program. If the function is dynamically liked we cannot analyze the other function.

4.1.3.3 *Handling of pointers*

The handling of pointers is only partially supported by our analysis. On the one hand we detect when the stack allocation of the pointer variable is accessed, meaning we notice that something was read from or written to the location behind the pointer. Furthermore, we can trace if a location is forwarded to another pointer, for example, passed to a function. But on the other hand we cannot fully follow the location, because it could change with pointer arithmetic during run-time. Furthermore, if a computation or control-flow decision is based on the value behind the pointer we cannot infer anything. The results of this analysis could be partially correct by adding dynamic run-time information about the pointers, but theoretically this problem undecidable.

4.2 EVALUATION OF OUR INTERACTION ANALYSIS

In this section, we show how we visualize the analysis structure to understand control flow between different regions. Furthermore, we give an overview of the test cases and we explain two test cases in more detail, using our visualization.

4.2.1 *Visualizing interaction analysis*

In order to better understand and debug our analysis we also implemented special `GraphTraits` to draw a CFG annotated with the information of a `FunctionNode`. The `InteractionAnalysisCFG` shows in each node, corresponding to a `BBNode`, the three sets IN, OwnAccesses, and OUT separated by – – –. Each set contains the `Accesses` displayed with the triple (Type | variable, RegionID). Figure 6 shows an example graph for the function `main`, the corresponding C++/IR code can be found in appendix A in Listings 19, 20. In the example graph, we see a write `Access` in the "if.then" block, that is propagated to the following BBs. This `Access` generates an interaction when the value gets used in the block "if.then.3" of Figure 6. If we compare the identified interaction with the code in Listing 19 we see that it corresponds to the read of the variable *a*, in the feature `BAR` that was previously written in the feature `FOO`. The automatic graph generation allows us to inspect the results and the state of our analysis, which eases debugging and later allows the user to examine interactions.

4.2.2 *Evaluating the interaction analysis*

In the previous section we presented our interaction analysis and how we automatically create graphs to ease debugging. We now evaluate if our analyses works as intended by verifying the generated graphs and comparing the results with our expectations. For this purpose we create different test cases that focus on specific cases like forwarding information to a function. All our test cases can be found in the VaRA repository at github[1]. This section gives an overview of our test cases and then describes the evaluation of two in more detail.

1 https://github.com/vulder/VaRA/tree/master/examples/FeatureDetection

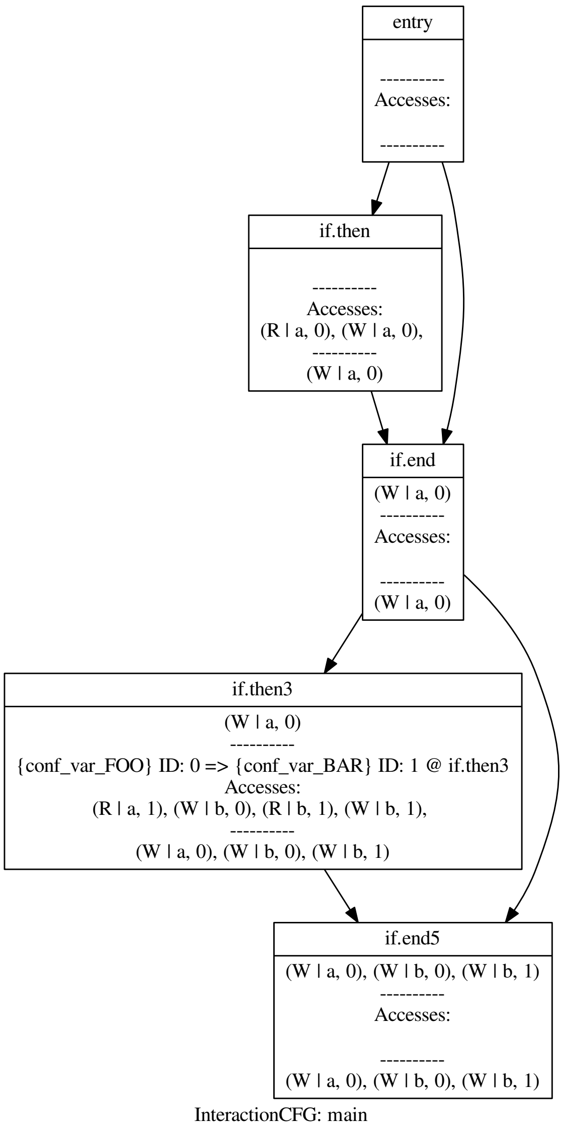


Figure 6: Interaction-analysis CFG for the function `main`.

4.2.2.1 *Test scenarios*

For the evaluation of our analysis we constructed different test scenarios where each focuses on a particular scenario our interaction analysis has to handle. The following list shows the different cases we extracted during development:

- Basic functionality:
  - simple data flow between two regions
  - multiple nested presence conditions
- Handling of function calls:
  - data flow between function using references/pointer semantics
  - data flow between function using copy by value semantics
  - data flow from value returned by a function
- Local relations:
  - blocking data flow with local overwrite
  - data flow through transitive relation
- Special cases:
  - data flow within a loop
  - data flow within recursive function
  - different levels of pointer indirection

For each test case, we create a small program that has some kind of data flow from one region to another, potentially causing an interaction. We check whether the created `Accesses` get propagate correctly through our graph and if the analysis can detect the interaction correctly. The first two test cases check the basic functionality; parsing presence conditions, creating `Accesses`, and propagating them through the graph. Then we check whether the analysis can handle function calls correctly and forwards all relevant `Accesses` to the IN set of the function, also testing if functions get reanalysed when their IN sets change. In addition, we also verify if we can detect an interaction in case a `Access` is returned from a function. Furthermore, we duplicate all our test cases to test value semantics as well as pointer/reference semantics, for example, when calling a function. Next we test if our analysis correctly deletes `Accesses` in case the variable is overwritten, meaning the previous `Access` is overwritten and the region would not influence the other region. Then we created test cases to check if a transitive `Access` correctly causes an interaction. In addition we also create examples for, some special cases, to handle loops and recursive function calls, which is important because in these cases we have cycles. Last we also created a larger test case that uses different levels of pointer indirection. Currently we correctly detect interactions in all our test cases, meaning we only find intended interaction.

We use the created test suite to ensure the correctness of our analysis and will expand it with further scenarios. In addition we also plan to evaluate our analysis on real world programs which currently cannot be done because our `clang` extension does not support all kinds of AST nodes.

4.2.2.2 *Testing interaction between different functions*

It is important for our analysis to correctly handle interactions between different functions. This means we have to detect an interaction in case that a value that was written within a feature region is used in another region within a different function. Figure 8 shows the interaction graph for the function `bazz`, from our test case. The code of the test case can be found in appendix A in Listing 21 and the graphs for the main function in Figure 7. At the beginning of the main function

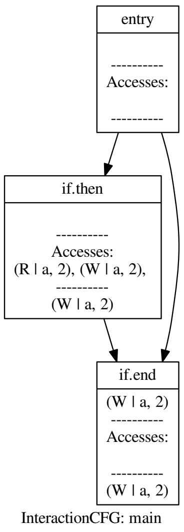
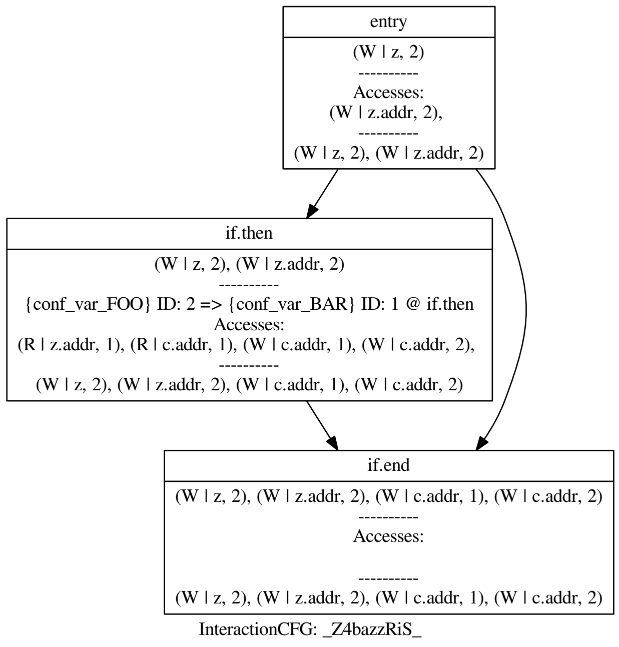


Figure 7: Interaction-analysis CFG for function `main`

the variable `a` is initialized and then incremented within the feature region `conf_var_FOO`, represented by the write `Access` in the "if.then" block of the main function, Figure 7. Later this variable is passed by reference to the two function `baz` and `bazz`. Within the function `bazz` the passed variable `a` is mapped first to the parameter `z` and then to the local reference allocation `z.addr`, therefore, we create a new `Access` to `z.addr`, representing the previous `Access` to `a`, and put it in the IN set of the `FunctionNode`. Later the variable `z.addr` is read, meaning the value of `a` influences some value within the feature region of `conf_var_BAR`. Hence, our analysis detect an interaction when then "if.then" block in graph, Figure 8, reads the `Access` to `z.addr`.


entry
(W | z, 2)

Accesses:
(W | z.addr, 2),

(W | z, 2), (W | z.addr, 2)
if.then
(W | z, 2), (W | z.addr, 2)

{conf_var_FOO} ID: 2 => {conf_var_BAR} ID: 1 @ if.then
Accesses:
(R | z.addr, 1), (R | c.addr, 1), (W | c.addr, 1), (W | c.addr, 2),
----
(W | z, 2), (W | z.addr, 2), (W | c.addr, 1), (W | c.addr, 2)
if.end
(W | z, 2), (W | z.addr, 2), (W | c.addr, 1), (W | c.addr, 2)

Accesses:

(W | z, 2), (W | z.addr, 2), (W | c.addr, 1), (W | c.addr, 2)
InteractionCFG: _Z4bazzRiS_

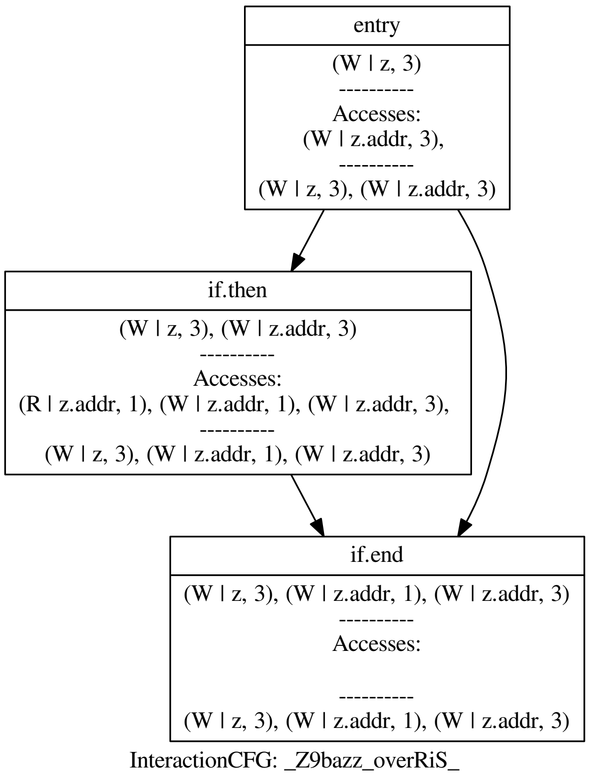
Figure 8: Interaction-analysis CFG for function `bazz`

Listing 18: TC: Overwriting previous `Access`.

```
1  void bazz_over(int &z, int &c) {
2    if (conf_var_BAR) {
3      z = 42; // detecting local overwrite
4      z = 2 + z;
5    }
6  }
```

4.2.2.3 *Testing overwriting accesses*

Another important case is overwriting an `Access`, this means, if a variable gets overwritten we have to delete the previous `Access` to it, because further accesses no longer get influenced by this `Access`. To test this we extend the previous example 21 with a new function `bazz_over`, shown in Listing 18. The difference between `bazz` and `bazz_over` is only in Line 3, here `42` is assigned to `a`, which leads to overwriting the previous access to `a` within feature `conf_var_FOO`. The rest of the program runs exactly as before, but if we now analyze `bazz_over` we should not find an interaction. In Figure 9, we show the `InteractionAnalysisCFG` for `bazz_over`. Our analysis detects the access as before and forwards a write `Access` to the IN set of function `bazz_over`, correctly transforming it to an `Access` to `z.addr`. However, although we have a read `Access` to `z.addr` withing the "if.then" block there is no interaction found. Hence, our analysis correctly identified an overwrite to `z` and invalidated the previous `Access`.


entry
(W | z, 3)

Accesses:
(W | z.addr, 3),

(W | z, 3), (W | z.addr, 3)
if.then
(W | z, 3), (W | z.addr, 3)

Accesses:
(R | z.addr, 1), (W | z.addr, 1), (W | z.addr, 3),

(W | z, 3), (W | z.addr, 1), (W | z.addr, 3)
if.end
(W | z, 3), (W | z.addr, 1), (W | z.addr, 3)

Accesses:

(W | z, 3), (W | z.addr, 1), (W | z.addr, 3)
InteractionCFG: _Z9bazz_overRiS_

Figure 9: `InteractionAnalysisCFG` for function `bazz_over`.

5 CONCLUDING REMARKS

We conclude the thesis by summarizing the presented framework VaRA and also our two analyses that detect features in LLVM-IR and find interactions between regions. Furthermore, we present future work, in which we describe how to solve existing problems with our analyses, and how we want to enhance our framework by providing even more support for researchers.

5.1 CONCLUSION

In this thesis, we have shown that the interfaces offered by VaRA enable the detection of specific code regions. In our example, we used the C/C++ frontend `clang` to extract code regions that correspond to software features and to annotate them in the intermediate representation using metadata. Then, we used the infrastructure of VaRA to create regions out of metadata that can be used by different analyses. Furthermore, we demonstrated that we could write an analysis that operates on these abstract regions. Our language-independent analysis could detect interactions among different regions. In our working example, we used VaRA to find feature interactions, information that could be used by developers to determine which features need to be tested together.

Supporting developers with tools and analyses to make debugging easier and software less error-prone is important as well as reducing work for researchers to develop new analyses. With VaRA, we have presented a framework that makes writing language-independent analyses easier and decouples them from the regions they work on. This allows us to run an analysis on different regions, enabling us to reuse analyses to investigate different problems. Furthermore, because we integrated VaRA into the LLVM compiler infrastructure, using the analyses inside the compiler or building tools gets easier.

5.2 FUTURE WORK

In Section 4.1.3, we mentioned some problems that arise regarding our interaction analysis. The first problem mentioned is a trade-off between the precision and the run-time of the analysis. We can either run the analysis context insensitively and consequently get more false-positives (meaning, we find interactions that are not present in any call context of the code base), or we analyze every function dependent on its call context (which makes our analysis very slow). This problem cannot be universally answered; scientists prefer accurate results and can handle longer analysis times, where compiler vendors need to keep the compile-time low, that is, an analysis has to be fast to be practical. Hence, we need to find a hybrid solution where we can tune the precision of the analysis. We plan to improve the analysis by making it context-sensitive. Additionally, we plan to reduce run-time by grouping similar call-contexts and by adding a threshold to limit the amount of contexts. This approach allows us to tune our analysis by selecting the amount of different call-contexts, either making it more precise or faster.

Another problem is that our analysis is designed as a module pass and can currently only process a single module at a time. We could address this problem in two ways: one solution is to do the analysis during link-time where we have the whole program there to analyze, but this means we have to reanalyze the complete module every time the IN set of a function within it changes, which could be very expensive. Another solution is to extend the current analysis to use further inputs beside the source code. We would analyze every module separately, but persist IN/OUT values to an information store from which the analysis can later obtain the values again, either in case it analyzes another module or in case it analyzes the same module again. For example, our analysis would store the OUT set of some function `bar` and, later, during the analysis of another module, use the stored information to handle analysis of the function call to `bar`.

Furthermore, we want to add dynamic informations to our analysis, such as call-contexts, to enhance the precision of our analysis, which can be combined with our second solution to the module-pass problem and provided from the same store. The information store could also contain run-time information about the program, such as pointer information, and provide it during analysis. For example, we annotate the program with measurement code that tracks the values of call parameters and stores this information. Then, during the next analysis of the program, we use this information to weigh which call-contexts should be analyzed. The information store acts as a cache and provides incrementally more information that could be gathered about the program, making the analysis more precise without increasing run-time significantly.

Another area where we see future work is extending the support VaRA offers to developers. We plan to add additional data structures, further analysis base classes (such as a `llvm::BasicBlock` pass) and we aim at enabling data sharing among analysis passes during optimization and analysis passes during link-time. In addition, we also want to provide generic analysis schemes, like a data-flow analysis, that can be specialized by researchers. For example, to create a data-flow analysis, a researcher would only have to specify which information is gathered and how different parts join their information. Furthermore, we are currently developing a way to analyze any arbitrary region within a program without implementing a new region type, allowing researchers to instrument the regions with markers that are automatically transformed into `IRegions`.

A APPENDIX: CODE

In this appendix, we show the extended source code to our small simplified examples.

A.1 CODE EXAMPLES FOR INTERACTION GRAPH.

Listing 19 shows the source code for our feature interaction example. The example has two code region that depend on the features (`conf_var_FOO` and `conf_var_BAR`). In the first region (`FOO`), `1` is added to the variable `a`. The variable `a` is later read within the other region (`BAR`), resulting in an interaction of the two features.

Listing 19: Example code for feature interaction between FOO and BAR.

```cpp
#include <cstdio>

volatile int conf_var_FOO = 1;
volatile int conf_var_BAR = 1;

int main(int argc, char *argv[])
{
  int a = 0;
  int b = 0;

  if (conf_var_FOO) {
    a += 1;
  }

  b += 1;

  if (conf_var_BAR) {
    b = a + 1;
  }


  printf("%d\n", a);
  printf("%d\n", b);
  return 0;
}
```

Listing 20 shows the complete LLVM-IR code that is generated for the code in Listing 19. In contrast to our previous LLVM-IR examples, Listing 20 is not simplified to reduce the burden on the reader and make it easier to understand. For our other examples, we removed details like `target datalayout` and various attribute annotations, because they are not important to understand our work. However, we include

Listing 20: IR code for the example shown in Listing 19

```
; ModuleID = 'SmallFlowExample.cpp'
source_filename = "SmallFlowExample.cpp"
target datalayout = "e-m:e-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-unknown-linux-gnu"

@conf_var_FOO = global i32 1, align 4
@conf_var_BAR = global i32 1, align 4
@.str = private unnamed_addr constant [4 x i8] c"%d\0A\00", align 1

; Function Attrs: norecurse uwtable
define i32 @main(i32 %argc, i8** %argv) #0 {
entry:
  %retval = alloca i32, align 4
  %argc.addr = alloca i32, align 4
  %argv.addr = alloca i8**, align 8
  %a = alloca i32, align 4
  %b = alloca i32, align 4
  store i32 0, i32* %retval, align 4
  store i32 %argc, i32* %argc.addr, align 4
  store i8** %argv, i8*** %argv.addr, align 8
  store i32 0, i32* %a, align 4
  store i32 0, i32* %b, align 4
  %0 = load volatile i32, i32* @conf_var_FOO, align 4
  %tobool = icmp ne i32 %0, 0
  br i1 %tobool, label %if.then, label %if.end, !Feature !1

if.then:                                          ; preds = %entry
  %1 = load i32, i32* %a, align 4
  %add = add nsw i32 %1, 1
  store i32 %add, i32* %a, align 4
  br label %if.end, !Feature !2

if.end:                                           ; preds = %if.then, %entry
  %2 = load i32, i32* %b, align 4
  %add1 = add nsw i32 %2, 1
  store i32 %add1, i32* %b, align 4
  %3 = load volatile i32, i32* @conf_var_BAR, align 4
  %tobool2 = icmp ne i32 %3, 0
  br i1 %tobool2, label %if.then3, label %if.end5, !Feature !3

if.then3:                                         ; preds = %if.end
  %4 = load i32, i32* %a, align 4
  %add4 = add nsw i32 %4, 1
  store i32 %add4, i32* %b, align 4
  br label %if.end5, !Feature !4

if.end5:                                          ; preds = %if.then3, %if.end
  %5 = load i32, i32* %a, align 4
  %call = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x i8], [4 x i8]* @.str, i32 0, i32
        0), i32 %5)
  %6 = load i32, i32* %b, align 4
  %call6 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x i8], [4 x i8]* @.str, i32 0, i32
         0), i32 %6)
  ret i32 0
}

declare i32 @printf(i8*, ...) #1

attributes #0 = { norecurse uwtable "correctly-rounded-divide-sqrt-fp-math"="false" "disable-tail-calls"
       ="false" "less-precise-fpmad"="false" "no-frame-pointer-elim"="true" "no-frame-pointer-elim-non-
       leaf" "no-infs-fp-math"="false" "no-jump-tables"="false" "no-nans-fp-math"="false" "no-signed-
       zeros-fp-math"="false" "no-trapping-math"="false" "stack-protector-buffer-size"="8" "target-cpu"=
       "x86-64" "target-features"="+fxsr,+mmx,+sse,+sse2,+x87" "unsafe-fp-math"="false" "use-soft-float"
       ="false" }
attributes #1 = { "correctly-rounded-divide-sqrt-fp-math"="false" "disable-tail-calls"="false" "less-
       precise-fpmad"="false" "no-frame-pointer-elim"="true" "no-frame-pointer-elim-non-leaf" "no-infs-
       fp-math"="false" "no-nans-fp-math"="false" "no-signed-zeros-fp-math"="false" "no-trapping-math"="
       false" "stack-protector-buffer-size"="8" "target-cpu"="x86-64" "target-features"="+fxsr,+mmx,+sse
       ,+sse2,+x87" "unsafe-fp-math"="false" "use-soft-float"="false" }

!llvm.ident = !{!0}

!0 = !{!"clang version 4.0.0 (git@github.com:vulder/vara-clang.git 526
       e9d51d2dfb8699c02faa767393123cd063ecd) (git@github.com:vulder/vara-llvm.git
       f791d9d8c5e28c2f9be5d3b3a55eba4fa829a93a)"}
!1 = !{!"[H,{conf_var_FOO}]"}
!2 = !{!"[T,{conf_var_FOO}]"}
!3 = !{!"[H,{conf_var_BAR}]"}
!4 = !{!"[T,{conf_var_BAR}]"}
```

one complete example in the appendix to show the full extend of an LLVM-IR file.

A.2 CODE EXAMPLES FOR TEST CASES

Listing 21 shows an example test case we use to test our feature extraction and interaction analysis. This test case in particular checks if our analysis correctly forwards the collected information to other function calls. All other test cases can be found in our repository[1].

Listing 21: TC: Interaction between different functions.

```
#include <cstdio>

volatile int conf_var_COMP = 1;
volatile int conf_var_FAST = 1;
volatile int conf_var_RECU = 1;
volatile int conf_var_FOO = 1;
volatile int conf_var_BAR = 1;

void baz(int &a, int &c) {
  if (conf_var_BAR) {
    c = a + 1;
  }
}

void bazz(int &z, int &c) {
  if (conf_var_BAR) {
    c = z + 1;
  }
}

int main(int argc, char *argv[])
{
  int a = 0;
  int b = 0;

  if (conf_var_FOO) {
    a += 1;
  }

  b += 1;

  baz(a, b);

  bazz(a, b);

  printf("%d\n", a);
  printf("%d\n", b);
  return 0;
}
```

1 https://github.com/vulder/VaRA/tree/master/examples/FeatureDetection

BIBLIOGRAPHY

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2Nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., 2006. ISBN: 0-321-48681-1.

[2] Florian Angerer, Andreas Grimmer, Herbert Prähofer, and Paul Grünbacher. "Configuration-Aware Change Impact Analysis." In: *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*. 2015, pp. 385–395.

[3] Sven Apel, Don S. Batory, Christian Kästner, and Gunter Saake. *Feature-Oriented Software Product Lines - Concepts and Implementation*. Springer, 2013. ISBN: 978-3-642-37520-0.

[4] Thorsten Berger, Steven She, Rafael Lotufo, Andrzej Wasowski, and Krzysztof Czarnecki. "Variability modeling in the real: a perspective from the operating systems domain." In: *ASE 2010, 25th IEEE/ACM International Conference on Automated Software Engineering, Antwerp, Belgium, September 20-24, 2010*. 2010, pp. 73–82.

[5] Janet Feigenspan, Christian Kästner, Sven Apel, Jörg Liebig, Michael Schulze, Raimund Dachselt, Maria Papendieck, Thomas Leich, and Gunter Saake. "Do background colors improve program comprehension in the #ifdef hell?" In: *Empirical Software Engineering* 18.4 (2013), pp. 699–745.

[6] Arnaud Hubaux, Yingfei Xiong, and Krzysztof Czarnecki. "A user survey of configuration challenges in Linux and eCos." In: *Sixth International Workshop on Variability Modelling of Software-Intensive Systems, Leipzig, Germany, January 25-27, 2012. Proceedings*. 2012, pp. 149–155.

[7] LLVM. *LLVM Language Reference Manual*. 2015. URL: `http://llvm.org/docs/LangRef.html` (visited on 01/16/2016).

[8] Chris Lattner. "LLVM: An Infrastructure for Multi-Stage Optimization." *See* `http://llvm.cs.uiuc.edu`. MA thesis. Computer Science Dept., University of Illinois at Urbana-Champaign, 2002.

[9] Chris Lattner. *LLVM*. 2015. URL: `http://www.aosabook.org/en/llvm.html` (visited on 01/16/2016).

[10] Jörg Liebig, Alexander von Rhein, Christian Kästner, Sven Apel, Jens Dörre, and Christian Lengauer. "Scalable analysis of variable software." In: *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Founda-*

tions of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013. 2013, pp. 81–91.

[11] ThanhVu Nguyen, Ugur Koc, Javran Cheng, Jeffrey S. Foster, and Adam A. Porter. "iGen: dynamic interaction inference for configurable software." In: *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*. 2016, pp. 655–665.

[12] Armstrong Nhlabatsi, Robin Laney, and Bashar Nuseibeh. "Feature interaction: the security threat from within software systems." In: *Progress in Informatics* 5 (), pp. 75–89.

[13] Steven She, Rafael Lotufo, Thorsten Berger, Andrzej Wasowski, and Krzysztof Czarnecki. "The Variability Model of The Linux Kernel." In: *Fourth International Workshop on Variability Modelling of Software-Intensive Systems, Linz, Austria, January 27-29, 2010. Proceedings*. 2010, pp. 45–51.

[14] Norbert Siegmund, Sergiy S. Kolesnikov, Christian Kästner, Sven Apel, Don S. Batory, Marko Rosenmüller, and Gunter Saake. "Predicting performance via automated feature-interaction detection." In: *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*. 2012, pp. 167–177.

[15] Norbert Siegmund, Alexander Grebhahn, Sven Apel, and Christian Kästner. "Performance-influence models for highly configurable systems." In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*. 2015, pp. 284–294.

[16] Tianyin Xu, Long Jin, Xuepeng Fan, Yuanyuan Zhou, Shankar Pasupathy, and Rukma Talwadker. "Hey, you have given me too many knobs!: understanding and dealing with over-designed configuration in system software." In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*. 2015, pp. 307–319.

DECLARATION

Hiermit versichere ich, dass ich diese Masterarbeit selbsständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe und alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, als solche gekennzeichnet sind, sowie dass ich diese Masterarbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt habe.

Passau, Germany, March 20, 2017

Florian Sattler