FirmSmith
Test Generation for Compiler Optimizations

Bachelor's Thesis of

Jeff Wagner

at the Department of Informatics
Institute for Program Structures and Data Organization (IPD)

Reviewer:	Prof. Gregor Snelting
Second reviewer:	Prof. Jörg Henkel
Advisor:	Andreas Zwinkau

1. October 2016 – 31. January 2017

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

I declare that I have developed and written the enclosed thesis completely by myself, and have not used sources or means without declaration in the text.
Karlsruhe, 30.01.2017

. . . . . .
(Jeff Wagner)

Abstract

Compilers transform intermediate representations of source code using target machine independent optimizations to improve the performance of the resulting machine code as much as possible, but are at the same time confronted with the requirement of producing correct code. Therefore, exhaustive testing is mandatory for ensuring the quality and correctness of compilers.

In this thesis, we present a random test case generator named FirmSmith for libFirm, which is an implementation of the intermediate representation FIRM that was developed at the University of Karlsruhe. Our test case generator improves the testing of libFirm, target independent optimizations by directly constructing test cases in FIRM and striving for covering a maximum of language semantic combinations. Thereby we are able to bypass the constraints on producible semantics a libFirm-based compiler front end could empose.

This thesis concludes that our test case generator produces a competitive coverage of optimizations, which is similar to relying on a libFirm-based compiler an existing test case generator. During the evaluation we compared FirmSmith to Csmith and determined that out of 37 tested optimizations, FirmSmith was able to trigger bugs in 12 and Csmith only in 6 optimizations.

Zusammenfassung

Compiler verbessern die Effizienz des erzeugten Maschinencodes indem sie zielarchitekturunabhängige Optimierungen auf einer Zwischendarstellung des Quelltextes anwenden. Gleichzeitig zur Anforderung Optimierungen immer effizienter zu gestalten, wird auch auch von Compilern erwartet, dass sie jederzeit korrekten Maschinencode erstellen.

In dieser Arbeit, präsentieren wir unseren Generator für zufällige Testfälle namens FirmSmith den wir für libFirm, eine Implementierung der an der Universität Karlsruhe entwickelten graphenbasierten Zwischensprache FIRM. Ziel unseres Testgenerator ist es, die Testabdeckung von libFirm zu verbessern und mögliche Fehler in der Implementierung aufzudecken. FirmSmith erstellt Programme direkt in der FIRM Darstellung und versucht eine möglichst hohe Anzahl an Kombinationen unterschiedlicher FIRM Semantik zu realisieren. Dadurch, dass FirmSmith direkt Graphen in FIRM erstellt, werden Einschränkungen durch das Compiler-Frontend bei der Erstellung von FIRM Graphen umgangen.

Die Schlussfolgerung dieser Arbeit stellt fest, dass wir eine Abdeckung der Optimierungen erreichen, die vergleichbar mit derer ist, die man mit bestehenden Testgeneratoren erreichen kann. Zudem stellen wir jedoch weiter fest, dass FirmSmith uns erlaubt Bugs in Optimierung zu finden, die Csmith entweder aufgrund der Struktur der Testfälle gefunden hat oder durch Einschränken des verwendeten Compiler-Frontends unmöglich finden konnte. Aus insgesamt 37 getesteten Optimierungen hat FirmSmith Bugs in 12 und Csmith in 6 Optimierungen gefunden.

Contents

1 Introduction

Compilers transform code written in high-level languages into semantic equivalent machine-level languages through analysis and subsequent transformations. First, the compiler uses syntax and semantical analysis to convert the source code into an intermediate representation, which is independent from the source language and may be modified in an optimization phase. Finally, the transformation phase of the intermediate representation produces the representation in the target language. [1]

FIRM [11] is a graph-based intermediate representation that was developed for compilers and the research thereof at the University of Karlsruhe. The library libFirm [4] implements FIRM and provides an interface to be used in compilers to create FIRM graphs representing the source code, to optimize FIRM graphs and to derive code for different target machines from them. [1]

Software, which is not written in assembler code for a specific target machine, relies on the correct implementation of compilers to ensure its correctness. Unfortunately, the process of source code analysis, optimization and code generation in compilers cause an increase of complexity and the likelihood of faults and wrong behavior. Therefore the implementation of compilers and their components has to be subject to rigorous testing to guarantee a maximum of reliability. In addition to hand-crafted test cases to detect bugs in compilers, random test case generators can be used to find bugs, as they can produce hundreds or thousands of bug inducing test cases. The University of Utah displayed the efficiency of test case generators with the inception of Csmith, a generator producing random C programs to test C compilers, which was able to lead to the discovery of more than 325 previously undetected bugs across multiple compilers. [12]

In order to ensure the FIRM implementation's quality and performance and to mitigate regressions in newer versions, the libFirm team maintains a flexible test suite. The test suite uses the libFirm-based C compiler Cparser to compare the processing of a collection of C programs known to be problematic with expected results and behavior, and thereby

the implementation of libFirm is indirectly tested. Existing random test case generators producing C programs, have been used by the libFirm team to create some of the test cases for the suite and to uncover bugs in their implementation.

The main goal of this thesis is to create a new test case generator to improve the quality of libFirm. The secondary objective is to increase durableness of bug inducing test cases. Instead of relying on compiler front ends to generate the FIRM graphs from a source language, we directly create FIRM graphs to test libFirm's optimization and code generation routines. The hypothesis is that generating the FIRM graphs directly achieves a higher coverage of libFirm by relying on features not currently used by compiler front ends and that test cases triggering libFirm bugs are unaffected by compiler front end changes.

2 Background

Compilers process code written in a programming language and translate it to a semantically equivalent representation, which is executable on a specific target machine. The compilation process may be split into three different parts, namely the front, middle and back end. First, the front end analyzes the source code and converts it to an intermediate representation (IR). Subsequently the IR may be subjected to an optimization phase in which the middle end performs target machine independent optimizations on the IR. Finally, the back end generates the code for the target machine from the IR emitted by the middle end. [1]

2.1 FIRM

The intermediate representation results from the analysis of source code, and must be able to describe the different semantics found in one or multiple source languages. The semantics in objective and imperative languages can be modeled using control and data flow, which express constructs such as conditionals, loops, branches, method calls and data dependencies. The IR is also the subject of optimizations in the compiler's middle end. Therefore the data structure implementing the IR must allow to efficiently perform program optimizations. The IR must furthermore be resembling the target machine code, as it is converted by the code generating back end of compiler software into the target language. [5].

FIRM is an intermediate representation for object oriented and imperative languages that was developed for compilers and the research thereof at the University of Karlsruhe. The name FIRM stands for Fiasco's intermediate representation mesh, as it was inititally conceived as part of the Fiasco compiler before its implementation was extracted into the open source library libFirm.

FIRM organizes the code representation hierarchical. At the top of the hierarchy is the representation of the entire program with all its available methods, types and entities. Methods describe the control and data flow using explicit dependency graphs, which are

directed, marked graphs. [4] The nodes have ordered inputs and outputs and an associated operation. The edges represent either control flow, data flow or memory dependencies and thereby form reversed flow graphs. An example of a complete FIRM graph is displayed in Figure 2.1.

2.1.1 Control Flow

The control flow of a function describes the order in which instructions of a program are executed. Graphs modeling the control flow are referred to as control flow graphs (CFGs). The nodes in a CFG represent basic blocks. Basic blocks are implicitly defined by the source code and contain the representation of instructions, which are bound to be executed before the control flow is changed. Therefore branches directing the control flow from one basic block to another must only be conducted after all other non-branching operations have been performed. The edges in a CFG represent the transfer of control induced by these branches. Each CFG consists of at least one entry and one exit block, which identify the start and end of a function's control flow. Entry blocks must not have any incoming edges and exit blocks must not have any outgoing edges. [1]
As FIRM uses dependencies to express the relation between different control flow blocks, its dependency graph equals a reversed control flow graph, where the dependency edges point into the opposite direction of the flow edges. Furthermore FIRM refers to Entry and Exit nodes as Start and End blocks. In order to traverse the control flow blocks in FIRM, the End block is used as a starting point for the traversals of the predecessors. Blocks which are unreachable by the traversal will be removed during dead code elimination. In order to preserve endless loops that prevent the flow from reaching the end node, an edge referred to as keep alive edge has to be introduced to connect the end node with a node belonging to the loop.

2.1.2 Data Flow

FIRM's data representation adheres to the Static Single-Assignment (SSA) property, which ensures that every variable in the program is only assigned once, in order for variable uses to always relate to exactly one definition. As values of variables might depend on the control flow, a special Phi-function is introduced by SSA, which references values inside the predecessor control flow blocks. Reverting a data flow graph yields a data dependency graph, in which the data dependency edges are in the opposite director of the flow edges. Every node in the data dependency graph represents a value and references the nodes representing the values it depends on. Due to the SSA-form the reference to

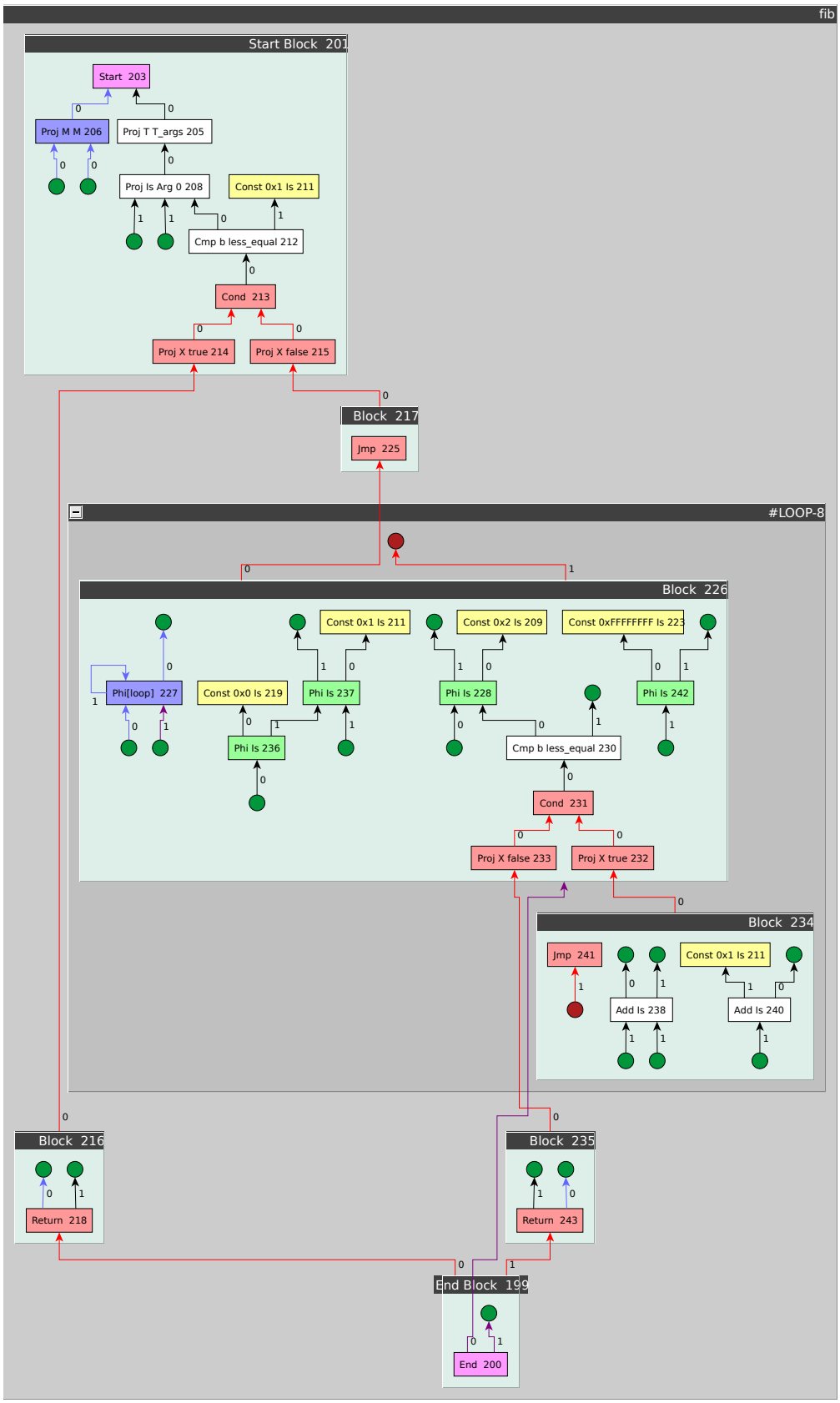

Figure 2.1: FIRM graph of method calculating Fibonacci number

```
1  int isPrime(int x) {
2    int r = 1;                    // (Entry)
3    int i = 2;                    // (Entry)
4    while (i < x && r == 1) {     // (a)
5      if (x % i == 0) {           // (b)
6        r = 0;                    // (c)
7      }
8      i++;                        // (d)
9    }
10   return r;                     // (e)
11 }                               // (Exit)
```

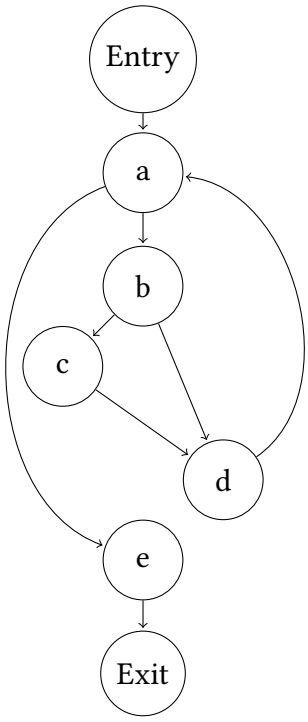

Figure 2.2: This figure shows a sample C function, which determines whether x is prime and its corresponding control flow graph. The *entry* node takes care of the variable initialization performed in line 1 and 2. Block *a* represents the loop header, which directs the control flow either to node *b*, the body of the loop, or to *e* the next block after the loop. Block *b* represents the If-statement which leads to *c* if the condition is fulfilled or otherwise to *d*, where the loop counter is increased and the flow is redirected to the loop header. After *c* modified the result variable, the flow is also directed to *d*. From *e* the *exit* node is reached and the function call is ended.

the dependency is unambiguous and unique, in contrast to non-SSA-form, where multiple definitions of a variable might exist.

FIRM nodes are always associated with predefined operations, which can be divided in two groups. The first group represents constructs as found on target machines, e.g. constants, comparisons, branches, algebraic or memory operations, whereas the second group relates to FIRM internal logic allowing to model blocks, synchronization, memory and tuples. Throughout this paper, nodes associated with an operation named *op* will be referred to as *op*-node.

Nodes consume different amounts of inputs and outputs depending on the associated operation. The output is always a single value, but may be a tuple to combine multiple output values. Proj-nodes can be used to extract individual results from tuples. As FIRM adheres to the Static Single-Assignment property, which ensures that every variable in the program is only assigned once, uses of variables always relate to exactly one definition, which can be referenced by the node. Nodes can directly reference their dependencies as they have exactly one definition.

2.1.3 Graph Uniformity

FIRM's explicit dependency graphs allow to describe both the control and data flow of methods in a single uniform graph. In order to achieve uniformity, control flow blocks are represented by Block-nodes. All nodes referring to non-block operations belonging to a block have the Block-node as input. Thereby the nodes in a block only become executable once the corresponding Block-node becomes available. The Block-nodes themselves have branching operations as input and become executable once one of possibly multiple inputs are available. The branch nodes are only marked executable once all data dependencies have been resolved.

2.1.4 Modes

Modes in FIRM relate to data types, which can be directly mapped to a representation on the target machine, e.g. integers, floating point types and pointers. FIRM also has special boolean and memory modes, which are used as the result of comparisons and memory operations respectively.

All FIRM nodes are typed with modes and have requirements on their inputs' modes. The nodes representing a certain operation may have a fixed modes or modes deduced from

the inputs. The restrictions on allowed input modes and the deduction rules to determine a node's mode specify how different nodes of different operations can be connected with each other.

2.1.5 Types

The type representation in FIRM is closely linked together with the code representation and gives information about the types' memory layout, their size and composition.

Primitive types are made up of integers, floating points and characters and are always associated with a mode that specifies how the type will be represented on the target machine. Compound types such as structs and unions are composed of possibly multiple members, which form a named relation between the compound type and the member type it encloses. Method types represent the types of methods, functions and procedures. They contain a list of the parameter and result types, as these are part of the type description Furthermore there are also types for modeling arrays.

2.2 Random Testing

Test case generators rely on different techniques to create test cases. Random test case generators create random inputs for the programs or functions to be tested[8]. grammar-based test case generators create input by randomly applying productions of a grammar and genetic test case generators repeatedly mutate a corpus of inputs and incorporate mutations triggering interesting or new behavior in the test subject.

2.2.1 Existing Test Case Generators

libFuzzer The LLVM Project [10] aggregates a set of compiler technologies and is most famous for the LLVM Core and the clang compiler. Both technologies revolve around an intermediate representation (IR) of program code. The LLVM Core provides architecture independent optimizations, which can be applied on the IR, as well as a code generation backend to translate the IR to assembly for the target architecture. The clang compiler is a frontend to translate C(++) and Object-C code to the IR.

In January 2015 the LLVM Project introduced libFuzzer [9], a tool to randomly test its subprojects. libFuzzer is an instrumenting and generative fuzzer for LLVM components. It uses a set of input files, referred to as corpus, as a starting point and mutates them

to test the components. Mutations which increase the code coverage are included in the the corpus and will be used to derive from in future mutations in order to strive for maximum code coverage. The testing of the components is made possible by the creation of entrypoints, which take the random input from the fuzzer and use it in the interaction with APIs. The random input can also be generated by other fuzz engines, such as AFL[13] or Radamsa[7]. Furthermore libFuzzer may be supplied with a dictionary of input related keywords to be used in the generation of mutations. It also supports the detection of CMP instructions and mutations based on their operands, as well as sanitizers to uncover memory leaks.

Multiple entrypoints have been created for libFuzzer to independently fuzz the different LLVM components, such as the clang compiler and LLVM's assembly and machine code processing implementations.

Csmith Csmith[12] is a randomized-test generation tool developed at the University of Utah. It creates random C programs, which cover most of the language's features. Unspecified behavior is avoided by Csmith in order to ensure that compilations of same C program with different compilers always produce the same output. The determinism of the output binaries allow to use differential-testing, which compares the output of the executables generated by different compilers, and thereby to discover wrong-code bugs. The random C programs generated by Csmith were used to test many different C compilers and discovered over 325 previously unknown bugs.

KTH The random-test case generator developed at the KTH Royal Institute of Technology in Stockholm as part of the thesis "Random Testing of Code Generation in Compilers" uses a grammar-based approach to create an intermediate representation to the test LLVM's code generation. [3]

3 Design

In this chapter, we first detail the design goals and the limitations to the scope of the thesis. test case generator FirmSmith. Afterwards we present our grammar based approaches to generate both control and data flow for FirmSmith's test cases.

FirmSmith's main objective is to enable the detection of previously undetected bugs in libFirm, which are either impossible or very unlikely to be found by relying on specific libFirm-based compilers. It should also serve as an exhaustive tool allowing to test new library features even before they are being used by compilers. Furthermore should FirmSmith be able to repeatedely produce indentical FIRM graphs and be unaffected by changes in libFirm-based compilers. Otherwise test cases could result in different FIRM graphs, which do not trigger the same behavior and bug in libFirm.

In the pursuit of the above mentioned goals, we chose to let FirmSmith directly build FIRM graphs instead of relying on the transformation of source code by front end. Thereby we can exhaust all of the available FIRM semantics are theoretically able to create every possible graph.

In order to reduce the extent of this thesis, we partially exclude FIRM semantics in the generation of test cases. We limit ourselves to the generation of reducible control flow graphs and a linear memory dependency model. Furthermore we do not generate test cases to check the handling of exceptions and we focus on the testing of libFirm's middle end, which performs target independent optimizations. The testing of the library's handling of invalid FIRM graphs is omitted and out of scope for FirmSmith.

3.1 Control Flow

The advantage of a graph based representation of a function's control flow is that its mathematical structure allows to formalize the flow and transformations applied upon it. Hecht[6] gives a formalization of a flow graph in Definition 1, which will be used subsequently as basis to define transformations.

Definition 1 *A flow graph is a triple $G = (N, E, n_0)$, where:*
(1) N is a finite set of nodes.
(2) E is a subset of N x N called the edges. The edge (n_1, n_2) enters node n_2 and leaves node n_1. We say that n_1 is a predecessor of n_2, and n_2 is a successor of n_1. A path from n_1 to n_k is a sequence of nodes $(n_1, .., n_k)$ such that (n_i, n_{i+1}) is in E for $1 < i < k$. The path length of $(n_1, .., n_k)$ is $k - 1$. If $n_1 = n_k$, the path is a cycle.
(3) n_0 in N is the initial node. There is a path from n_0 to every node.

3.1.1 Reducibility

CFGs of goto-less programs possess the reducibility characteristic, as defined by Allen [2]. Hecht[6] proved that these CFGs may be reduced to a single node by iteratively applying two simple graph transformations T1 and T2, as formalized in Definition 2 and Definition 3, in an arbitrary order.

Definition 2 *Let $G = (N, E, n_0)$ be a flow graph. Let (n, n) be an edge of G. Transformation T1 is the removal of this edge.*

Definition 3 *Let n_2 not be the initial node and have a single predecessor, n_1. Transformation T2 is the replacement of n_1, n_2 and (n_1, n_2) by a single node n. Predecessors of n_1 become predecessors of n. Successors of n_1 or n_2 become successors of n. There is an edge (n, n) if and only if there was formerly an edge (n_2, n_1) or (n_1, n_1).*

T1 may be applied on a node n in the CFG, which has a self-loop, an edge pointing to itself, T1 removes that edge, whereas T2, if applied to a node n_2 is merged together with its single predecessor n_1, into a unified node n. The resulting node n has exactly one edge to the successors of the previous nodes n_1 and n_2. Figure 3.1 shows an example of how the application of the transformations T1 and T2 can be combined to reduce a control flow graph to a single node.

3.1.2 Inversed Reducibility

The reducibility characteristic is useful for the generation of random control flow graphs, as demonstrated by Hansson[3], inasmuch as that if we are inversing the process of reducing the graph into a single node, we can generate any arbitrary reducible graph by applying graph transformations on an initial graph containing a single node. As any reducible graph can be reduced into a single node, the transformations T1 and T2 cannot possibly be injective and the reverse images projected by the inversed transformations $T1^{-1}$ and/or

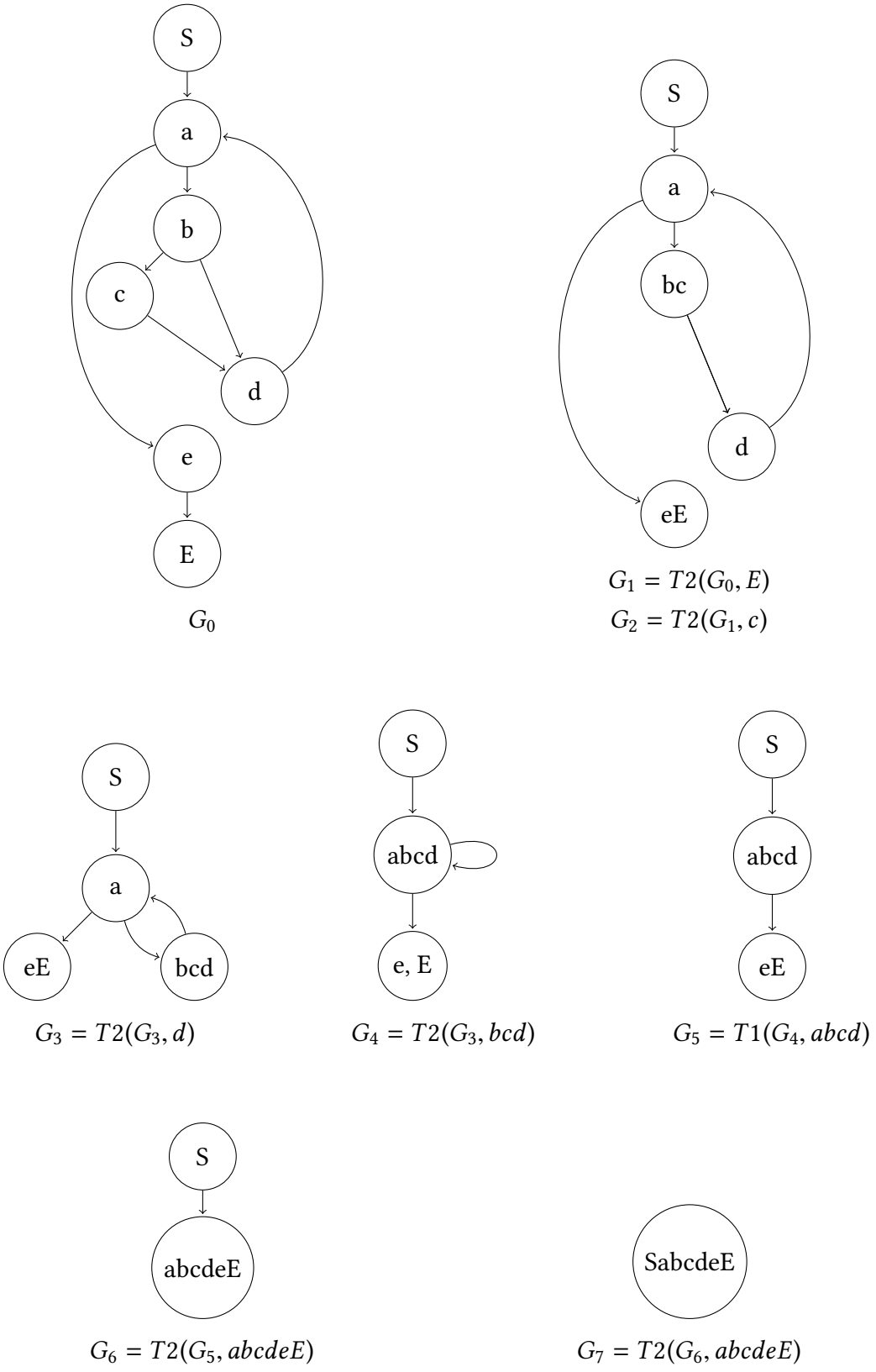

Figure 3.1: Reducing the control flow graph of prime-testing method presented in Figure 2.2 using the transformations specified in Definition 2 and Definition 3.

$T2^{-1}$ must be a set with multiple graphs. As a matter of convenience, we will denote the inversed transformations $T1^{-1}$ and $T2^{-1}$ as $I1$ and $I2$ respectively.

Applying $I1$ to a node n results in adding the edge (n, n) and underlies the condition that the edge (n, n) does not already exist, $I2$ if applied on a node n with l successors, adds a new node n_new. Every successor of n may result in either a successor of n_1, n_2 or both. As there are these three possible choices for the successors, the image of $I2$ contains of 3^l different graphs. $I2$ must not be applied on the start node in order to ensure that we always have a valid start node.

Definition 4 *Let $G = (N, E, n_0)$ be a flow graph. Let n be a node of G where $(n, n) \notin E$. Then the transformation $I1$ adds the edge (n, n).*

$$I1(G, n) = (N, E \cup (n, n), n_0)$$

Definition 5 *Let $G = (N, E, n_0)$ be a flow graph and $n \neq n_0 \in N$ with l successors $\{s_0, \ldots, s_{l-1}\}$ and $E_{ns} = \{(n, x) \mid x \in N)\}$ the set of edges connecting the node n with its successors. Then we define the transformation $I2$ as adding a node n_{new} as follows:*

$$I2(G, n) = \{I2_w(G, n) \mid w \in \{0, 1, 2\}^l\}$$
$$I2_w(G, n) = (N \cup \{n_{new}\}, m(w), n_0)$$

where $m : \{0, 1, 2\}^l \to P(N \times N)$, $l \in \mathbb{N}$ and $e : \{0, 1, 2\} \to P(N \times N)$ are defined as:

$$m(w) = (E \setminus E_{ns}) \cup \left(\bigcup_{0 \leq i < |w|} e(s_i, w_i) \right)$$

and

$$e(z, x) = \begin{cases} \{(n, x)\} & \text{if } z = 0 \\ \{(n_{new}, x)\} & \text{if } z = 1 \\ \{(n, x), (n_{new}, x)\} & \text{if } z = 2 \end{cases}$$

3.1.3 Graph Construction

FirmSmith constructs the control flow graph by applying a random sequence of the transformations presented in section 3.1.2 until the necessary number of blocks is reached.

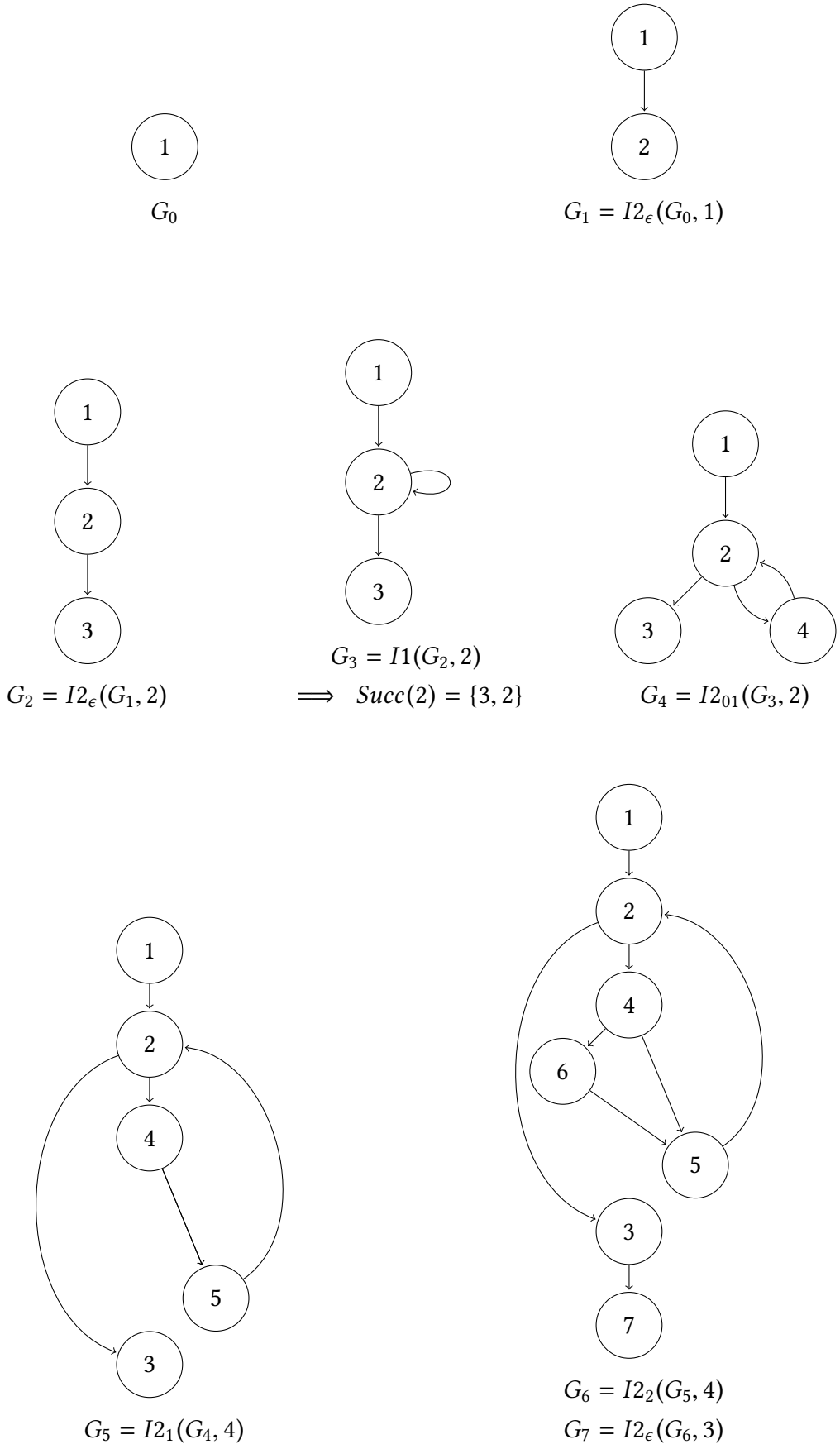

Figure 3.2: Construction of the control flow graph of prime-testing method presented in Figure 2.2 using the transformations specified in Definition 4 and Definition 5.

The blocks have forward edges pointing to their successors as well as backward edges to their predecessors in order to allow both top-down and bottom-up traversal of the graph.

The construction of the control flow graph is succeeded by the conversion to a FIRM graph by introducing Block-nodes and branch nodes. Conditional branch nodes depend on temporary placeholder values, whose replacement is discusses in section 3.2.

3.2 Data Flow

FirmSmith iteratively builds the FIRM data dependency graph. First we create placeholders for FIRM nodes and link them to control flow blocks. The data flow graph is gradually expanded by replacing these placeholders by more complex expressions. The operands to these expressions might themselves be placeholders. Thereby we are able to expand the data flow graph to a desirable size.

As variables in SSA-form are always unambiguously defined by the result of a single operation, FIRM mostly omits the notion of variables. FIRM nodes simply represent an operation, the result thereof and the mode of the result. The operands of these operations are other nodes, representing other operands. FirmSmith reintroduces the notion of variables as combinations of a FIRM node and a FIRM type. Furthermore, these variables inherit the properties of FIRM nodes, in particular, the SSA-property. They are also always linked to a control flow block.

FirmSmith uses dummy variables as placeholders for future data. Dummy variables are variables with a fixed type and a Dummy-node. In subsequent steps, the Dummy-node can be replaced by any FIRM operation, which supports the variable's type. We refer to the process of replacing Dummy-nodes as resolving a variable.

Replacing a variable either creates new dummy variable, which need to be resolved, in the same CFB or in case of a Phi-node in the predecessor CFBs. Therefore we traverse the blocks in the CFG from bottom to top and resolve the variables for every CFB. When replacing a dummy variable and fill in an actual FIRM node, we must to consider the mode of the FIRM node. Hence we categorize the strategies to resolve variables depending on their ability to replace a FIRM node with a certain mode. Operand consuming operations expand the size of the graph and are chosen with decreasing probability, as the graph is reaching the desired size. On the contrary, operations which consume no operands limit the size and chosen with increasing probability. The probability of choosing semantics

that are expanding the graph decrease as the graph grows in order to force the number of variables to converge to zero.

3.3 FIRM Types

Supporting multiple types in FirmSmith is essential to create diverse randomized programs and to stress a higher portion of the FIRM implementation. Therefore we generate random FIRM types to be used in our test case generation. The construction of these random types is strated by creating an initial set of primitive types, which will be used to compose compound types. The set of generated types may then be used in the construction of random method types.

3.4 Bug Classification

Random test generators are able to always create new test cases, which trigger software bugs, as long as they run. Inspecting hundreds or thousands of failed test cases is unfeasible and inefficient for a human operator. Therefore we need to reduce the amount of test cases to be evaluated and to classify the test cases in groups relevant to a specific bug together, so that only test cases the most favorable to analyses may be inspected in order to find the fault.

Software bugs may manifest as different symptoms, crashes, early abortions, wrong results or non-termination. [12] The manifested symptoms are used to group the failure inducing test cases into different categories. We work under the presumption, that different bugs show different symptoms, and that the same bugs show the same symptoms. This represents a trade-off, as the symptom resulting from one bug may hide another and multiple bugs may indeed result in the same symptom, especially in case of abortions due to constraint violations in the software.

In order to isolate the effects of bugs from each other, we reduce the covered by each compiler run. We investigate specific optimizations one by one and thereby prevent bugs in previously run optimizations to affect subsequently run ones. Furthermore we analyze the assertion and verification messages, as well as the stacktrace to classify bugs.

4 Implementation

This chapter describes the implementation details our test case generator and gives an overview over the construction of test cases.

We decided to separate the test runner from the actual test case generartion, which we made the sole concern of FirmSmith. Therefore FirmSmith's only task is the generation of random FIRM graphs. Its responsibility is to accept a seed for its Pseudorandom number generator and options for constraining the FIRM graph. In order to test libFirm with random generated graphs, we created a *fuzzer*, which takes the output from a random test case generator and uses it as an input to a program interfacing libFirm's optimization and code generation functionality.

4.1 Control Flow

As FIRM requires that every graph has at least a Start and an End block, we first create a CFG with two connected nodes. In order to expand the graph, we randomly choose a node from the current version of the CFG and pick either the transformation $I1$ or $I2$. If the node does violate the constraints underlying the application of the chosen transformation, we start over again and choose a new node and a new transformation. Otherwise the graph is enlarged by applying the chosen transformation to the chose node.

The transformation $I1$ does not increase the number of the nodes, but does increase the amount of edges. $I1$ can always be applied to the nodes of the CFG, if they have not already a self-loop. As a measure to ensure that the CFG has always one start block and at least one exit block, we disallow applying $I1$ to either the initial start block or to exit blocks.

The transformation $I2$ does increase the number of nodes exactly by one and the number of edges at least by one. Therefore in order for our CFG to reach a designated amount of c nodes to ensure a certain size, we need to apply $I2$ exactly $c - 2$ times to the initial version of the CFG. $I2$ may also create new exit blocks, if the node it is applied on has no

```
CFBs = [startBlock, endBlock] ;
for i = 0; i ≤ c - 2; i++ do
    N = NULL ;
    while N == NULL do
        T = pick random ∈ {I1, I2} ;
        B = pick random ∈ CFBs ;
        if T appliable on B then
            if T == I1 then
                I1(B)
            else
                N = I2(B) ;
                B.append(N) ;
            end
        end
    end
end
```

Figure 4.1: Construction graphs using the transformations detailed in Definition 4 and Definition 5

successors or none of them are chosen to be added to the new node. The algorithm for the generation of the control flow graph is displayed in Figure 4.1.

Once the CFG construction is completed, we need to translate it from FirmSmith's internal representation to FIRM's representation, which requires us to translate the nodes in the CFG to FIRM blocks and create FIRM branching operations inside these blocks and register them as predecessors to the blocks corresponding to our nodes. Therefore we traverse the nodes in the CFG beginning at the start node, create blocks for every successor node, create the branching operations for the block belonging to the currently traversed node, and add these operations as predecessor to the blocks belonging to the successor nodes. This traversal results in a new FIRM graph, where the nodes are defined by blocks and the edges by the blocks pointing to the branching operation in the predecessor. Blocks always require at least as many branching operations as they have successor nodes. The references from blocks to the branching operations in their predecessors correspond to the edges in our CFG. Therefore for a block with a single successor, we create an unconditional jump operation. For a block with two successors, we create two Proj-nodes depending on

a boolean condition, for which we add a dummy variable to the graph. For more than two successors, we use a switch to determine to which successor to redirect the control flow. The different switch cases depend on an integer, for which we add a corresponding dummy variable. If a block has no successor, we encounter a possible a exit of the functions an create a Return-node and dummy variables for the different return values.

4.2 Type Generation

Primitive as well as random compound types to be used in the test case are created during initialization process and is based on three subsequent steps.

The first step is to create primitive types, which all other randomized types will be based on. We create primitive types for numbers, booleans and pointers and add them to a type pool. This pool contains all the types available to the type generation algorithm.

After the initial pool of primitive types has been established, we create the desired amount of compound types and randomly pick either a struct or union type to be created. The compound type may be composed of an arbitrary number of types which are then referred to as entities of the compound type. The entities' types may either be already defined types, pointers to these, or pointers to the enclosing type, thereby forming a recursive type. Beside the type information, the entities also store the offset of that type in the compound type, so that in a lowering phase, the correct pointer arithmetic can be deduced. After the compound type was constructed, it is added to the pool of available types and can therefore be included as an entity by following compound type constructions.

In a third an final step, we create a given number of method types for the functions we want to generate. The method types describe the prototype of a method and thereby the types of the parameters and results. The returning of multiple result values is supported, but must also be made possible by the used calling convention. The different types referred to by the method types are randomly picked from the pool. Method types are not added to the previously filled type pool, as we do not support the passing of functions as parameters or results.

4.3 Data Flow

The generation of the data flow relies on the incremental replacement of placeholders as already explained in section 3.2. Additional to the set of dummy variable created during the

Operation	Number	Pointer	Boolean
Constant	✓		
Algebraic	✓		
Convert	✓		
Minus	✓		
Memory Allocation		✓	
Memory Load		✓	
Member		✓	
Compare			✓
Existing	✓	✓	
Mux	✓	✓	✓
Phi	✓	✓	✓
Function call	✓	✓	✓

Figure 4.2: Table shows the supported operations for the different kinds of variables

control flow graph conversion described in section 4.1, we add already resolved variables to the start block to represent parameters passed to the function.

For every control flow block the replacement of a dummy variable is subdivided in three steps. First, a resolver class is chosen based on the dummy variable's type. Three resolver classes exist for distinguishing the replacement of numbers, pointers and booleans, which are listed in Figure 4.2 together with the strategies they support. Second, the probabilities for each replacement strategy are calculated. Finally, a replacement strategy is picked according to the computed probability distribution. If the dummy variable's type is not supporting the chosen strategy, a new one is randomly picked using the same distribution. The probability for each replacement strategy depends on an interpolation of a start and end weight using the ratio of the actual block size to the desired size as parameter. The weights can individually be defined for every strategy. After the weights have been calculated the probability of choosing a strategy equals the ratio of its weight to the total weight.

Preserving SSA-form If we replace a variable with an existing FIRM node, we must ensure that the node does not directly or indirectly depend on the the variable's uses. This is in particular important when replacing dummy variables with already resolved variables. We determine whether a variable depends on another by traversing all its predecessors inside the control flow block. If we encounter the dummy variable during the traversal of the predecessors, the variable depends on the dummy variable and cannot be used as replacement. Figure shows a valid replacement of a graph with dummy variables followed by a replacement violating the SSA-property.

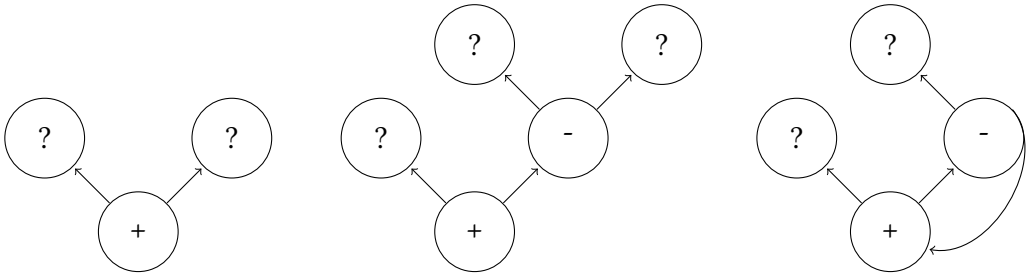

Figure 4.3: SSA-form violating replacement. First an operand of the Add-node with two dummy variables is resolved by placing a Sub-node and introducing two new dummy variables. Second, an operand for the Minus-node is replaced by the existing Add-node, which is an SSA-violation as the Add-node already depends on the Minus-operand.

5 Evaluation

This chapter describes the evaluation of our implementation through inspecting and comparing the efficiency FirmSmith's generated test case to those of Csmith. First, we introduce the evaluation setup followed by the results of the evaluation and a discussion thereof.

5.1 Fuzzer

In order to methodically run the generated test cases against libFirm, we developped a generic fuzzer that allows to switch between multiple FIRM graph producing test case generators, as well as between multiple libFirm entry points. Therefore we seperated the fuzzer conceptually in a front and a backend as shown Figure 5.1. The frontend runs the test case generators and provides the test case. The test case is then used as an input to test the different optimizations exposed by the backend. The backend wraps the programs serving as entry points to the libFirm implementation. In order to isolate bugs stemming from the optimization functions, the backend should allow to perform optimizations individually. Furthermore the fuzzer is a able to create reports for failed or crashing backend executions and assists with categorizing the bugs.

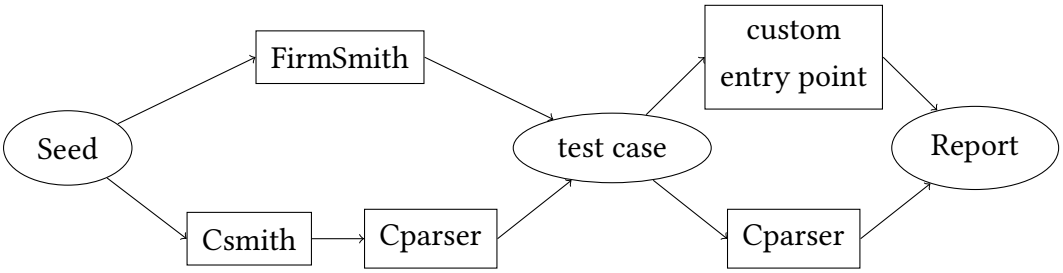

Figure 5.1: Architecture of test runner

For our test setup we created two frontends to run FirmSmith and Csmith respectively. As Csmith produces random C programs and not FIRM graphs, the Csmith frontend relies on Cparser to convert the C program to a FIRM graph while ensuring that Cparser does

not already perform any optimizations on the test case before it is passed to the backend. Furthermore we wrote two backends to wrap both CParser and a custom entry point. The custom entry point allows to directly run libFirm's optimization functions on the FIRM graphs represented in the test cases.

5.2 Results

We tested 37 different optimizations, of which 6 and 12 crashed when confronted with test cases genereated by the Csmith and FirmSmith frontend respectively. However the number of optimizations does not directly correlate with the number of distinct bugs, as the optimizations depend on shared code to gather information and to transform the the graph. Only further manual analysis can help to clarify, whether multiple crashes are related. For example, our fuzzer's bug classifier grouped the several crashes found in the *local*, *deconv*, *if-conversion* and *shape-blocks* together, because they failed with the verification message, that a block was unreachable.

Optimization	Csmith	FirmSmith
combo		✓
deconv		✓
gcse		✓
gvn-pre	✓	
if-conversion		✓
local		✓
lower-mux		✓
parallelize-mem	✓	✓
reassociation	✓	✓
remove-phi-cycles	✓	✓
shape-blocks	✓	✓
target-lowering		✓
thread-jumps	✓	✓

Figure 5.2: Optimization bugs discovered by FirmSmith and Csmith

5.2.1 Coverage

We use both FirmSmith and Csmith to generate 500 test cases each and collected information about which code was covered in libFirm. We focused on analysing the code coverage in libFirm's optimization subfolder in order to analyze which code is run and what FIRM graph constructions are not generated by FirmSmith. The code coverage is displayed in Figure 5.3.

File	Csmith	FirmSmith
boolopt.c	46.8%	24.8%
cfopt.c	88.0%	94.9%
code_placement.c	96.2%	96.2%
combo.c	89.0%	91.1%
convopt.c	94.3%	92.2%
critical_edges.c	82.4%	82.4%
dead_code_elimination.c	100.0%	100.0%
funccall.c	69.7%	69.7%
garbage_collect.c	94.6%	68.9%
gvn_pre.c	95.7%	97.1%
ifconv.c	86.7%	89.3%
instrument.c	0.0%	0.0%
ircomplib.c	56.2%	56.2%
irgopt.c	99.0%	99.0%
iropt.c	69.2%	72.3%
iropt_t.h	100.0%	100.0%
jumpthreading.c	80.9%	86.4%

File	Csmith	FirmSmith
ldstopt.c	73.4%	58.2%
loop.c	23.4%	56.1%
occult_const.c	100.0%	100.0%
opt_blocks.c	95.0%	94.8%
opt_confirms.c	2.3%	2.3%
opt_frame.c	100.0%	17.2%
opt_inline.c	76.3%	40.0%
opt_osr.c	84.9%	49.3%
parallelize_mem.c	100.0%	95.3%
proc_cloning.c	25.1%	24.7%
reassoc.c	76.9%	84.7%
return.c	93.1%	93.1%
rm_bads.c	33.3%	63.5%
rm_tuples.c	92.0%	92.0%
scalar_replace.c	87.2%	7.3%
tailrec.c	16.1%	89.6%
unreachable.c	81.1%	81.1%

Figure 5.3: Coverage report of libFirm's optimization implementation after running tests against both the FirmSmith and Cparser backend

FirmSmith realized *70%* a coverage, whereas Csmith realized *84%*. FirmSmith disadvantage considering the code coverage mainly lies in the lacking support for arrays, a more complex memory model and global variables. FirmSmith's advantage stems from the fact that it has more nesting and a more diverse combination of operations.

FirmSmith scores a lower code coverage in the *boolean optimization*, which to optimize boolean conditions by transforming logical combinations of boolean conditions into a

more efficient representation. FirmSmith does not produce these constructs, although it is theoretically possible to do so, due to the probability distribution used in the data flow generation. Furthermore FirmSmith does not induce as many optimizations in the *Load/Store optimization*, as it only allow linear memory dependencies.

5.3 Mux Failure

The Mux-node is a node available in FIRM to implement a multiplexer. It accepts a boolean selector and takes two other inputs. Depending on the selector's value being either false or true, the first or second input is passed as output respectively. Compilers could for example use the Mux-node to implement expressions or assignments using a ternary operator. Additionally libFirm converts if-conditions into Mux-nodes in the optimization phase.
During the fuzzing of the Mux lowering optimization, we encountered an assertion fail for almost every generated test case. The excerpt of the stacktrace in Figure 5.5 confirms that the failure is encountered, while the compiler is in the code generation phase and trying to lower the Mux-nodes. The error output displayed in Figure 5.4 together with the extended stacktrace shows that the failure is due to a node being a Proj-node instead of a Phi-node as expected.

```
Assertion failed: (is_Phi_(phi)), function set_Phi_next_, file ./ir/ir/
   irnode_t.h, line 671.
```

Figure 5.4: Failed assertion during Mux-node lowering

In the lowering phase FIRM replaces the Mux-node with a construction of conditional jumps and a Phi-node if Mux operations are not supported by the target architecture. First, the control flow block containing the Mux-node is split into a lower and an upper part, which are linked together with an unconditional jump from the upper to the lower block. The lower part contains the Mux-node and all its successor nodes, whereas the upper part contains all the predecessor nodes. Then the jump connecting the upper and lower block is made conditional on the Mux-node's selector node. The false jump directs the control flow to a newly created block, which is added as a predecessor to the lower block. The Mux-node is then replaced with a Phi-node. The Mux-node's inputs are then correctly selected by the Phi-node, depending on the control flow entering from the upper block or the false block.
Looking at the plot of the control block containing the Mux-node in Figure 5.6, we notice that the node's inputs refer to the same node. This usage is permitted by the API, but

```
set_Phi_next_ at irnode_t.h:671 ((ir_node *)phi=[432, Proj], (ir_node *)
set_Phi_next_ at irnode_t.h:671 ((ir_node *)phi=[432, Proj], (ir_node *)next
    =[427, Phi])
add_Block_phi_ at irnode_t.h:686 ((ir_node *)block=[521, Block], (ir_node *)
    phi=[432, Proj])
collect_new_phi_node at irgmod.c:115 ((ir_node *)node=[432, Proj])
lower_mux_node at lower_mux.c:73 ((ir_node *)mux=[433, Mux])
lower_mux at lower_mux.c:101 ((ir_graph *)irg, (lower_mux_callback *)cb_func
    )
do_lower_mux at firm_opt.c:334 ((ir_graph *)irg)
do_irg_opt at firm_opt.c:441 ((ir_graph *)irg, (const char *)name)
do_firm_optimizations at firm_opt.c:596 ()
optimize_lower_ir_prog at firm_opt.c:757 ()
generate_code at firm_opt.c:772 ((FILE *)out, (const char *)input_filename)
```

Figure 5.5: Stacktrace at the time of the assertion failure displayed in figure 5.4.

is rather rare as the usefulness of this construct is questionable. When we the lowering algorithm tries to create the Phi-node consuming these inputs, libfirm recognizes that the Phi-node is superfluous and that the input nodes can be directly used. Therefore the Phi creation API returns a reference to the Proj-node, instead of a Phi-node. Therefore the attempt to collect the Phi-node in a next step fails, and triggers the assertion failure.
Fuzzing libFirm with IR graphs generated by Cparser's frontend using random C programs did not result in triggering this failure, as the current version of Cparser makes no use of Mux-nodes to represent source code. The detection of this bug showcases FirmSmith's usefulness as bugs can be revealed in libFirm's functionality even before frontends start relying on it through automatic testing.

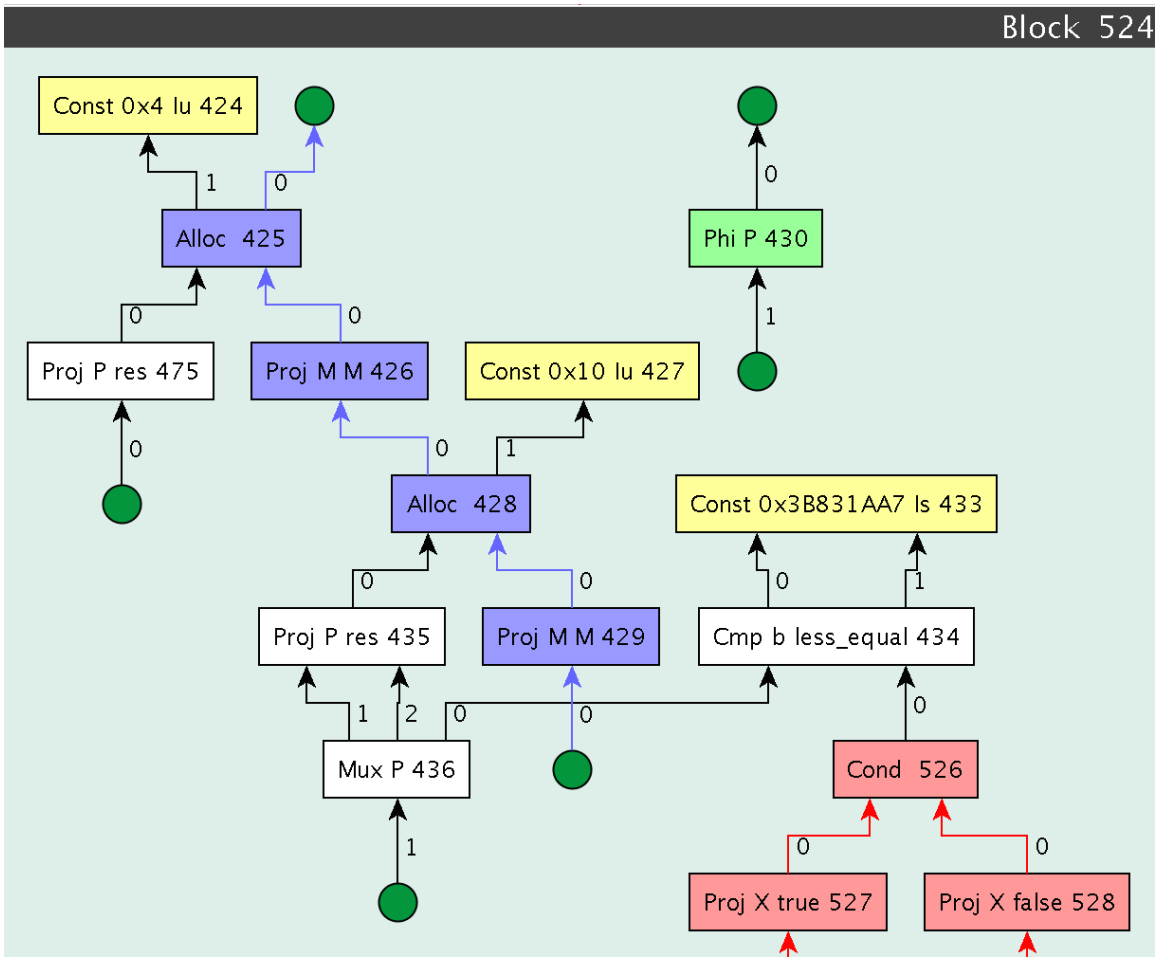

Figure 5.6: The block containing assertion fail provoking Mux-node.

6 Conclusion

We can conclude that FirmSmith achieves a competitive coverage of optimizations that is comparable to the use of Csmith. Furthermore, we were able to use FirmSmith to create test cases with bug triggering semantics, that would have been impossible to be found with Csmith.

6.1 Future Work

Additions to FirmSmith's graph generation algorithm could help to find new bugs, increase test coverage and an easement of analyzing bug triggering test cases.

6.1.1 Memory Dependency Graph

The memory dependence graph inside a control flow block is currently only a linear list, but FIRM allows to specify partial memory blocks, which allow to parallelize independent memory operations and to be merged for operations relying on multiple memory blocks. FirmSmith could be expanded to generate these partial memory blocks.

6.1.2 Feedback control

Currently the test case construction is solely governed by the provided seed and the specified parameters. Information about how the test case impacted the execution inside libFIRM is not considered by FirmSmith. FirmSmith could be extended by a feedback control, which could both be used to reduce the test case size as well as for increasing the invasiveness.

Test cases generated by FirmSmith might consist of thousands of nodes and despite a best-effort to ease the detection of the fault inducing bug, it requires a lot of manual effort to isolate the bug. A feedback loop could help FirmSmith to minify the test case, by methodically removing functions, blocks and nodes, which do not lead to the bug triggering.

The invasiveness by FirmSmith could be increased by feedback about which source code during the test was covered and which control path was chosen. Furthermore FirmSmith could try to avoid graph constructions which repeatedely lead to triggering already-detected bugs.

Bibliography

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1986. ISBN: 0-201-10088-6.

[2] Frances E. Allen. "Control Flow Analysis". In: *SIGPLAN Not.* 5.7 (July 1970), pp. 1–19. ISSN: 0362-1340. DOI: 10.1145/390013.808479. URL: http://doi.acm.org/10.1145/390013.808479.

[3] Hansson Bevin. *Random Testing of Code Generation in Compilers*. 2015.

[4] Sebastian Buchwald and Andreas Zwinkau. "Instruction Selection by Graph Transformation". In: *Proceedings of the 2010 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*. CASES '10. Scottsdale, Arizona, USA: ACM, 2010, pp. 31–40. ISBN: 978-1-60558-903-9. DOI: 10.1145/1878921.1878926. URL: http://doi.acm.org/10.1145/1878921.1878926.

[5] Ron Cytron et al. "Efficiently Computing Static Single Assignment Form and the Control Dependence Graph". In: *ACM Trans. Program. Lang. Syst.* 13.4 (Oct. 1991), pp. 451–490. ISSN: 0164-0925. DOI: 10.1145/115372.115320. URL: http://doi.acm.org/10.1145/115372.115320.

[6] M. S. Hecht and J. D. Ullman. "Characterizations of Reducible Flow Graphs". In: *J. ACM* 21.3 (July 1974), pp. 367–375. ISSN: 0004-5411. DOI: 10.1145/321832.321835. URL: http://doi.acm.org/10.1145/321832.321835.

[7] Aki Helin. *Radamsa - a general-purpose fuzzer*. https://github.com/aoh/radamsa.

[8] Roy P. Pargas, Mary Jean Harrold, and Robert R. Peck. "Test-Data Generation Using Genetic Algorithms". In: *Software Testing, Verification And Reliability* 9 (1999), pp. 263–282.

[9] LLVM Project. *libFuzzer – a library for coverage-guided fuzz testing*. http://llvm.org/docs/LibFuzzer.html.

[10] LLVM Project. *The LLVM Compiler Infrastructure*. http://llvm.org/.

[11] Martin Trapp, Götz Lindenmaier, and Boris Boesler. *Documentation of the Intermediate Representation FIRM*. Tech. rep. 1999-14. Universität Karlsruhe, Fakultät für Informatik, Dec. 1999. Chap. 3, pp. –40.

[12] Xuejun Yang et al. "Finding and Understanding Bugs in C Compilers". In: *SIGPLAN Not.* 46.6 (June 2011), pp. 283–294. ISSN: 0362-1340. DOI: 10.1145/1993316.1993532. URL: http://doi.acm.org/10.1145/1993316.1993532.

[13] Michal Zalewski. *american fuzzy lop*. http://lcamtuf.coredump.cx/afl/.