Masarykova univerzita
Fakulta informatiky

Extracting Parts of Programs into Separate Binaries

Master's thesis

Tomáš Mészaroš

Brno, 2018

Declaration

Hereby I declare, that this paper is my original authorial work, which I have worked out by my own. All sources, references and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Tomáš Mészaroš

Advisor: Mgr. Marek Grác, Ph.D.

Acknowledgement

I would like to thank my advisor Marek Grác for his time and for making this thesis possible. Many thanks to my consultant Viktor Malík for sharing his knowledge and willingness to always help. Thanks to Pavel Odvody for coming up with the thesis idea and for his valuable feedback.

A big thanks to my family, Alena, Ľudovít and Martin for their support during my studies.

I would like to thank everybody that supported me in any shape or form on this journey. In no particular order: Majo, Jozef, Paťo, Mikasa, Astrix, Miša, Markéta, Tomáš, Marcel, Janka, Hanka, Žofka, Vlastík, Mišo, Dominik, Stela, Ondra, Kristína, Grandma, Lukáš, Majky, Maroš, Palo and others that I forgot to mention. Thank you.

Abstract

This thesis presents method for extracting parts of programs into separate binaries. The extraction method is based on static analysis of the program source code and leverages the LLVM infrastructure. Solution is implemented as LLVM optimization pass and is integrated into the open-source tool APEX. Extracted binary can be executed repeatedly without any other intermediary manual steps. Final solution is tested on three UNIX utilities.

Keywords

static analysis, program analysis, program extraction, intermediate representation, code optimization, LLVM

Contents

1 Introduction

Running debugger with set breakpoint at the selected variable location is the usual approach when user wants to know value of the variable. Unfortunately, this approach is cumbersome when user wants to execute this procedure many times. It consists of many manual steps, which are time consuming to perform. Ideally, there should be a script that accepts line of code as an input and produces value of the selected variable once the execution hits the desired line of code.

Normally, this method would require to use debugger with the scripting support and write scripts that would instruct debugger what exactly to do, basically replicating the manual approach.

Instead of scripting debugger to do the extraction, we could write a tool that would accept line of code as an input, run analysis on where the execution path in the program occurs to get to the target instruction and transplant subset of the program with computed execution path into the separate binary. This way, user would have separate executable that, when executed, would produce value of the targeted instruction without having to manually step thought or script debugger.

This thesis aims to devise method for statically transplanting a subset of a C program and implement it in a open-source tool. The selected program subset should be extracted from the original program provided by the user and synthesized as an independent, executable binary.

Proposed solution should be implemented in a tool having appropriate form, preferably using LLVM infrastructure. It should be user friendly and allow user to provide input of choice.

Implemented tool should be tested on at least two real-world open-source C programs in order to find where the room for improvements is and what could be achieved in the future.

The remainder of this thesis is structured as follows.

In chapter 2 we briefly introduce the LLVM compiler infrastructure and explain what makes it so popular.

We present method that is able to extract part of programs in chapter 3, while chapter 4 explains specific implementation details.

Experiments and results are discussed in chapter chapter 5.

Finally, chapter 6 summarizes the results of this thesis and presents possible further research and development opportunities.

2 The LLVM Compiler Infrastructure:

The LLVM project is a collection of modular and reusable compiler and toolchain technologies. [LLV18h] Designed to be compatible with already existing UNIX tools, LLVM includes set of low-level tools like Clang compiler, assembler, debugger, etc. [Lat18]

2.1 Architecture

The main distinguishing feature that separates LLVM from other compiler frameworks is its internal structure. LLVM compiler leverages tree-phase architecture shown in Figure 2.1 to achieve high degree of flexibility and modularity.

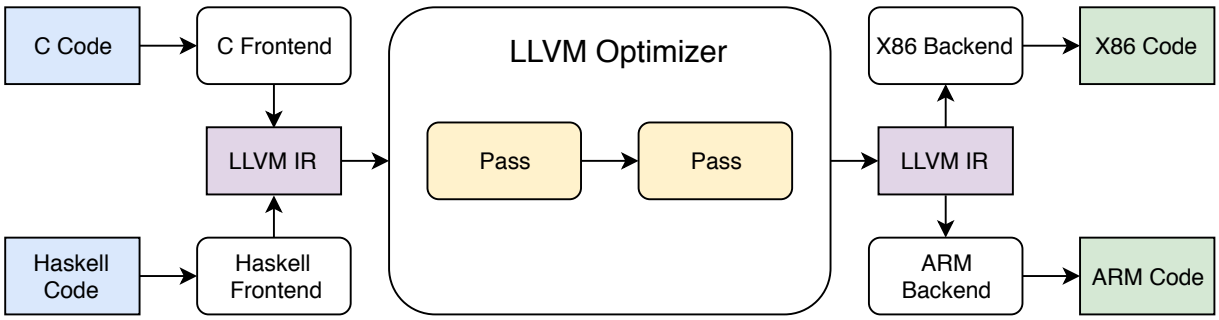

Figure 2.1: LLVM Compiler Architecture.[Lat18]

There are three distinct parts of the LLVM architecture show in Figure 2.1:

- Frontend is responsible for the first phase of compilation. Input code is parsed, processed, validated for errors and translated into the LLVM Intermediate Representation (IR). IR is used to represent code in the compiler (more about IR in section 2.2).
- IR code may be optionally put through optimization during the second phase. Optimizer can use various analysis and transform passes to improve the code and emit modified IR.
- Third and final phase is the code generation. Various backends take IR and produce platform specific machine code.

The huge advantage of this architecture is the fact that the optimizer and backend phase work with the IR instead of the language specific source code. This means that the compiler developer can write frontend for the new language that translates source code to the IR and does not have to write optimizations and code generator because LLVM infrastructure works with the IR and already provides those facilities.

2.2 Intermediate Representation

LLVM assembly language is a static single assignment (SSA)[1] *based representation that provides type safety, low-level operations, flexibility, and the capability of representing 'all' high-level languages cleanly. It is the common code representation used throughout all phases of the LLVM compilation strategy.[LLV18c]*

LLVM code can be represented by the following three equivalent forms:

1. Compiler intermediate representation (IR) that resides in the memory.
2. Bitcode stored in the file.
3. Assembly representation that is human readable.

We present the simple C function foo, stored in the file foo.c:

foo.c

```c
int foo(int a, int b) {
    if (a > b) {
        return a+b;
    } else {
        return a-b;
    }
}
```

The LLVM Intermediate Representation of the foo.c is available in the following code:

foo.s

```
define i32 @foo(i32 %a, i32 %b) #0 {
entry:
  %retval = alloca i32, align 4
  %a.addr = alloca i32, align 4
  %b.addr = alloca i32, align 4
  store i32 %a, i32* %a.addr, align 4
  store i32 %b, i32* %b.addr, align 4
  %0 = load i32, i32* %a.addr, align 4
  %1 = load i32, i32* %b.addr, align 4
  %cmp = icmp sgt i32 %0, %1
  br i1 %cmp, label %if.then, label %if.else

if.then:                                          ; preds = %entry
  %2 = load i32, i32* %a.addr, align 4
```

1. SSA means that each variable is defined before it is being used and is assigned exactly once.[Ras10]

```
  %3 = load i32, i32* %b.addr, align 4
  %add = add nsw i32 %2, %3
  store i32 %add, i32* %retval, align 4
  br label %return

if.else:                                          ; preds = %entry
  %4 = load i32, i32* %a.addr, align 4
  %5 = load i32, i32* %b.addr, align 4
  %sub = sub nsw i32 %4, %5
  store i32 %sub, i32* %retval, align 4
  br label %return

return:                                           ; preds = %if.else, %if.then
  %6 = load i32, i32* %retval, align 4
  ret i32 %6
}
```

LLVM programs are composed from units called **modules**. Two or more modules can be combined together with the linker.[2] Figure 2.2 shows architecture of the LLVM module.

Each module can include several functions, which in turn can contain several basic blocks.

Function in the LLVM defines a list of basic blocks, which form the control flow graph of the function (blocks A and B in Figure 2.2). [LLV18c]

A **basic block** is a type of a container that contains instructions that are executed sequentially[LLV18d]. Each basic blocks is formed with non-terminating instructions and a single terminator instruction at the end of the block. Only terminator instructions can terminate a basic block[3] (in the case of Figure 2.2, terminators would be Instruction 3 in both basics blocks).

In the case of `foo.s`, we can see that the function `foo` contains four basic blocks: `entry`, `if.then`, `if.else` and `return`.

Basic block `entry` is special in a way that it is the first basic block in the `foo` function and thus is executed always first. It is also prohibited to have any predecessors, unlike basic block `if.then`, which have `entry` as it predecessor and thus allow branching of the program.

2.3 Optimizations

LLVM uses the concept of passes for the optimizations. Concrete optimizations are implemented in a **pass** that works with some portion of program code (e.g. module, function,

2. Two bitcode modules can be linked with tool llvm-link. More info at https://llvm.org/docs/CommandGuide/llvm-link.html
3. https://llvm.org/doxygen/classllvm_1_1TerminatorInst.html

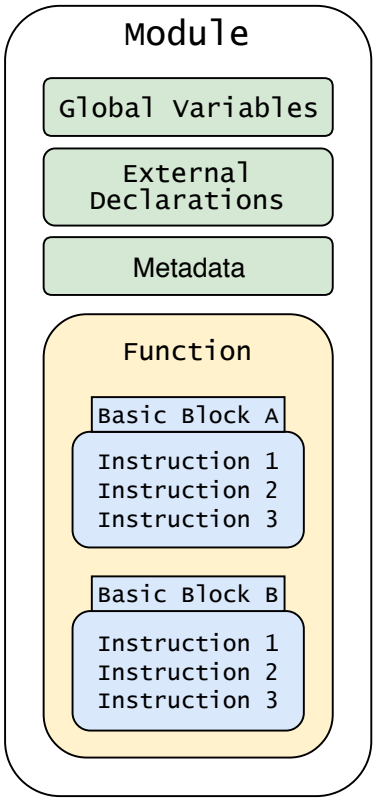

Figure 2.2: LLVM Module Structure.

loop, etc.) to collect or transform portion of the code[LLV18e]. Pass is a optimization unit that is executed in the second phase of the compilation process (Figure 2.1) or can be executed separately with opt tool.[4]

There are three types of passes:

1. **Analysis:** Collect information from the IR and feed it into the other passes. They can be also used for the debugging purposes (e.g. pass that counts number of functions in the module).
   Examples:
   - basiccg: Basic CallGraph Construction.
   - dot-callgraph: Print Call Graph to "dot" file.
   - instcount: Counts the various types of Instructions.
2. **Transform:** Change the program in some way. They can use data that was produced by analysis pass.
   Examples:
   - dce: Dead Code Elimination.
   - loop-deletion: Delete dead loops.
   - loop-unroll: Unroll loops.
3. **Utility:** Do not fit into analysis or transform pass categories.
   Examples:

4. https://llvm.org/docs/CommandGuide/opt.html

- verify: Module Verifier.
- view-cfg: View CFG of function.
- instnamer: Assign names to anonymous instructions.

2.4 Clang

The Clang project provides a language front-end and tooling infrastructure for languages in the C language family.[LLV18a]

Clang compiler uses LLVM infrastructure for optimizations and code generation (as is described in Figure 2.1). Clang AST (Abstract Syntax Tree)[5] closely represents the underlying code and does not abstract away elements that are useful refactoring tools[LLV18b].

add.c

```
int add(int a, int b) {
    return a+b;
}
```

Getting the AST from the code `add.c` is as simple as running clang with the following command line arguments:

```
clang -Xclang -ast-dump -fsyntax-only add.c
```

add.c AST

```
TranslationUnitDecl 0x622b600 <<invalid sloc>> <invalid sloc>
... leaving out internal clang declarations ...
‘-FunctionDecl 0x6280fa8 <ast.c:1:1, line:3:1> line:1:5 add ’int (int,
↪  int)’
  |-ParmVarDecl 0x622c268 <col:9, col:13> col:13 used a ’int’
  |-ParmVarDecl 0x6280ed0 <col:16, col:20> col:20 used b ’int’
  ‘-CompoundStmt 0x6281150 <col:23, line:3:1>
    ‘-ReturnStmt 0x6281138 <line:2:2, col:11>
      ‘-BinaryOperator 0x6281110 <col:9, col:11> ’int’ ’+’
        |-ImplicitCastExpr 0x62810e0 <col:9> ’int’ <LValueToRValue>
        | ‘-DeclRefExpr 0x6281090 <col:9> ’int’ lvalue ParmVar 0x622c268
        ↪  ’a’ ’int’
        ‘-ImplicitCastExpr 0x62810f8 <col:11> ’int’ <LValueToRValue>
          ‘-DeclRefExpr 0x62810b8 <col:11> ’int’ lvalue ParmVar 0x6280ed0
          ↪  ’b’ ’int’
```

5. Syntactic structure of the source code represented in as a tree.

Clang AST provides great value for developers. Nevertheless, instead of the AST, we will work in this thesis directly with the IR, because it provides greater flexibility via LLVM API.[6]

Clang is also directly able to emit IR for the `add.c` with the following command:

```
clang -S -emit-llvm ast.c -o ast.s
```

add.s

```
define i32 @add(i32 %a, i32 %b) #0 {
entry:
  %a.addr = alloca i32, align 4
  %b.addr = alloca i32, align 4
  store i32 %a, i32* %a.addr, align 4
  store i32 %b, i32* %b.addr, align 4
  %0 = load i32, i32* %a.addr, align 4
  %1 = load i32, i32* %b.addr, align 4
  %add = add nsw i32 %0, %1
  ret i32 %add
}
```

6. https://llvm.org/doxygen/namespacellvm.html

3 Extracting Program Subsets

In this chapter, we introduce a method for extracting chosen parts of programs into separate executable binaries.

First, we start with the method overview in section 3.1, where we define what the method input is and briefly outline the method itself - what it does and what are the steps for achieving the final result.

Next, we follow with an example of the method execution from the user perspective in section 3.2.

In the rest of the chapter, we present in detail each major step of the method.

3.1 Method Overview

The input of the proposed method consists of two main parts:

- **code**: A C program compiled into the LLMV bitcode.
- **target**: An integer value representing a line of code from the original C program.

We also define **source** to be an entry to the code (the `main` function).

The method determines what parts of the code to extract according to the source and the target. The procedure calculates a possible program path from the source to the target and extracts this program path into a separate, functioning executable.

Barring implementation specific details (which are discussed in chapter 4), the method can be summarized by the following five steps:

1. Compute data dependencies between instructions.
2. Find connected components in the computed data dependencies inside every function.
3. Construct a call graph - a mapping between connected components and functions that are being called from these components.
4. Find a path from the source to the target in the call graph.
5. Eliminate dead components and functions that do not depend on the path.

After the completion of the steps mentioned above, we denote the produced LLVM bitcode as **output**. This is defined as a part of the original program containing a path from the source to the target that is consistent (i.e. that can be compiled independently). Moreover, we expect the output to be executed the same way as the original.

3.2 Example

To better illustrate the explained methods, we give a running example. Let the code be the following C program shown in the `example.c`:

example.c

```c
int foo(int n) {
    int x = n + 10;
    return x;
}

int bar(void) {
    int y = 42;
    return y;
}

int main(void) {
    int some_int = 10;
    int foo_result = foo(some_int);
    int bar_result = bar();
    return 0;
}
```

Next, let the input target be the line 7. We define source to be the start of the main function. Since our method does not work directly with the C source code but instead works with the LLVM Intermediate Representation (IR), use clang compiler to translate the C code into IR:[1]

```
clang -S -emit-llvm example.c -o example.s
```

The emitted LLVM IR is shown in `example.s`:[2]

example.s

```
define i32 @foo(i32 %n) #0 {
entry:
```

1. Flag `-S` tells clang to only run preprocess and compilation steps, while `-emit-llvm` makes sure to use the LLVM representation for assembler and object files. For detailed description of various clang flags, visit https://clang.llvm.org/docs/ClangCommandLineReference.html.
2. Strictly speaking, this is not exactly the IR code that would be emitted by the clang. We have stripped out the module info and the comments to make it more readable. To see the unmodified `example.s`, please go to the Appendix B.

```
3   %n.addr = alloca i32, align 4
4   %x = alloca i32, align 4
5   store i32 %n, i32* %n.addr, align 4
6   %0 = load i32, i32* %n.addr, align 4
7   %add = add nsw i32 %0, 10
8   store i32 %add, i32* %x, align 4
9   %1 = load i32, i32* %x, align 4
10  ret i32 %1
11 }
12
13 define i32 @bar() #0 {
14 entry:
15  %y = alloca i32, align 4
16  store i32 42, i32* %y, align 4
17  %0 = load i32, i32* %y, align 4
18  ret i32 %0
19 }
20
21 define i32 @main() #0 {
22 entry:
23  %retval = alloca i32, align 4
24  %some_int = alloca i32, align 4
25  %foo_result = alloca i32, align 4
26  %bar_result = alloca i32, align 4
27  store i32 0, i32* %retval, align 4
28  store i32 10, i32* %some_int, align 4
29  %0 = load i32, i32* %some_int, align 4
30  %call = call i32 @foo(i32 %0)
31  store i32 %call, i32* %foo_result, align 4
32  %call1 = call i32 @bar()
33  store i32 %call1, i32* %bar_result, align 4
34  ret i32 0
35 }
```

The target line number 7 from the `example.c` corresponds to the line 16 from the `example.s`.

The procedure for finding mapping between C a code line and its analogous IR instruction is described in chapter 4.

When we apply the proposed method on the program in `example.s` with respect to the source and the given target, we get output as shown in `example_extracted.s`:

`example_extracted.s`

```
define i32 @bar() #0 {
entry:
```

```
  %y = alloca i32, align 4
  store i32 42, i32* %y, align 4
  ret i32 %0
}

define i32 @main() #0 {
entry:
  %bar_result = alloca i32, align 4
  %call1 = call i32 @bar()
  store i32 %call1, i32* %bar_result, align 4
  ret i32 0
}
```

As we can see from the `example.c`, the execution path from the program entry to the **target** does not include the function `foo` and its associated instructions. We are left only with the function `bar` which contains target and the necessary instructions in the function `main`.

The source given in `example_extracted.s` can be recompiled back into a functioning executable. This process is described in chapter 4.

3.3 Computing Data Dependencies

In order to identify what parts of the IR are safe to be removed, it is essential to compute dependencies between instructions. This helps us to preserve consistency of the remaining code so that it can be later compiled into a functional executable.

We recognize two types of dependencies between IR instructions: control and data dependencies. The following terminology and procedures for computing dependencies that we use are taken from the Marek Chalupa Master's thesis *Slicing of LLVM Bitcode* [CHA25].

- **Control dependence** describes what nodes are controlled by which predicate.
- **Data dependence** between nodes A and B exists if definitions from A reach uses in B.

From the perspective of our method, the important information comes from data dependencies.

We define **data dependency graph** $G = (V, E)$, where V is a set of vertices that correspond to instructions and E is a set of edges that correspond to data dependencies between instructions.

The following example demonstrates how data dependencies are used for the previously presented `example.c` source code. Taking closer look specifically at the function `main`:

```
define i32 @main() #0 {
entry:
  %retval = alloca i32, align 4
  %some_int = alloca i32, align 4
  %foo_result = alloca i32, align 4
  %bar_result = alloca i32, align 4
  store i32 0, i32* %retval, align 4
  store i32 10, i32* %some_int, align 4
  %0 = load i32, i32* %some_int, align 4
  %call = call i32 @foo(i32 %0)
  store i32 %call, i32* %foo_result, align 4
  %call1 = call i32 @bar()
  store i32 %call1, i32* %bar_result, align 4
  ret i32 0
}
```

In order to compute the data dependency graph, we use the `dg` library. [CHA25] [3] Computed data dependency graph for instructions from the function `main` is presented in the Figure 3.1. This graph is used in the next step of the method for finding connected components (section 3.4).

To illustrate how the graph can be used, we take a closer look at the instruction: `%some_int = alloca i32, align 4`. We can see that the following instructions have data dependency on the `%some_int`:

```
  store i32 10, i32* %some_int, align 4
  %0 = load i32, i32* %some_int, align 4
```

It is apparent that both instructions need `%some_int` for their operand. If we removed `%some_int = alloca i32, align 4` without taking into consideration that there are two instructions that depend on it, we would get into inconsistent state where two instructions would contain undefined values as their operand.[4] This would lead into unsuccessful recompilation of the modified code back into an executable.

3. For more info please visit https://github.com/mchalupa/dg [CHA25]. We will take a closer look at `dg` in the chapter 4.

4. More about undefined values at https://llvm.org/docs/LangRef.html#undefined-values, https://llvm.org/docs/FAQ.html#what-is-this-undef-thing-that-shows-up-in-my-code, https://llvm.org/doxygen/classllvm_1_1UndefValue.html

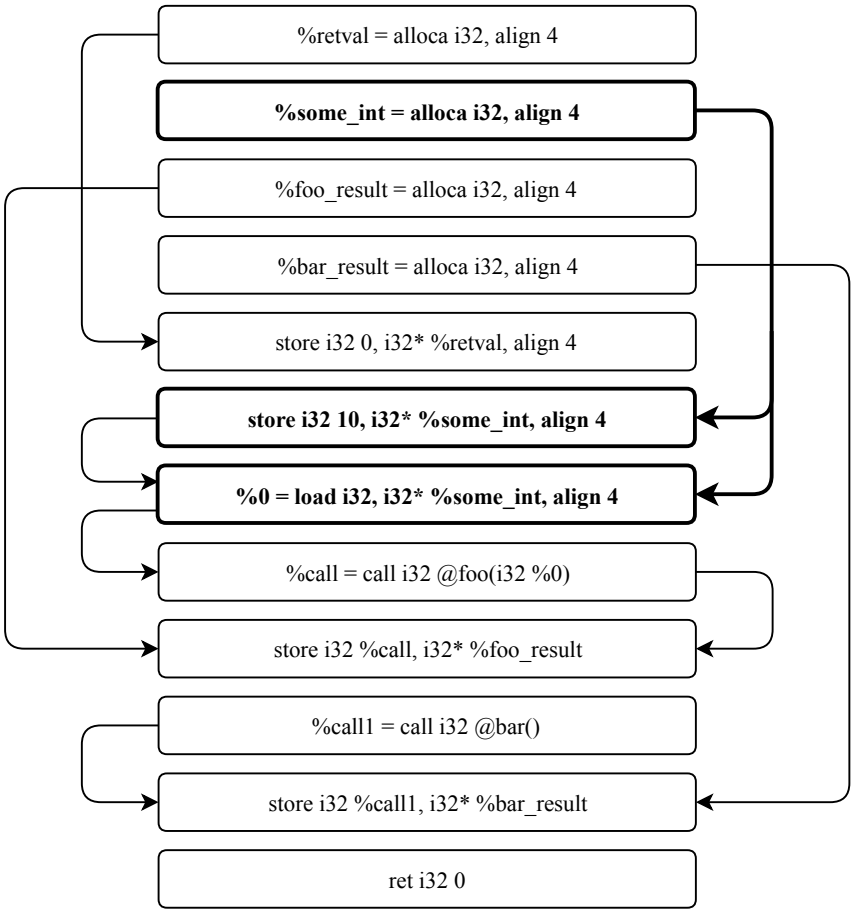

Figure 3.1: Data dependencies graph of the `main` function instructions.

3.4 Finding Connected Components

In the previous section, we defined the instruction data dependencies and we have shown how they can be used to preserve code consistency. However, they do not fully solve our problem of knowing when it is safe to remove instruction. Data dependencies between only two instructions do not reveal the possible the chain of dependencies that may form between multiple instructions. To resolve this problem, we propose the idea of computing connected components of the data dependency graph.

We define a **connected component** as an isolated subgraph, where each pair of vertices is connected by some path. For the purposes of computation, we treat this subgraph as it would have undirected edges.

We use the data dependency graph computed in section 3.3 and we find its connected components using the following algorithm:

finding components

```
0. Let G(F) = data dependency graph for function F
1. Let SF = set of all functions in code
```

```
2. FOR EACH function F in SF:
       FOR EACH instruction I in F:
           Run Breadth-first search on G(F), starting from I:
               Add all reachable/visited instruction to the component,
               ↪ unless
               they are already in some other component.
```

We run the above mentioned algorithm separately on the data dependency graph of each function in the input. As an example, we present components for `main` function in Figure 3.2 (for clarity, each component is distinguished by its own color).

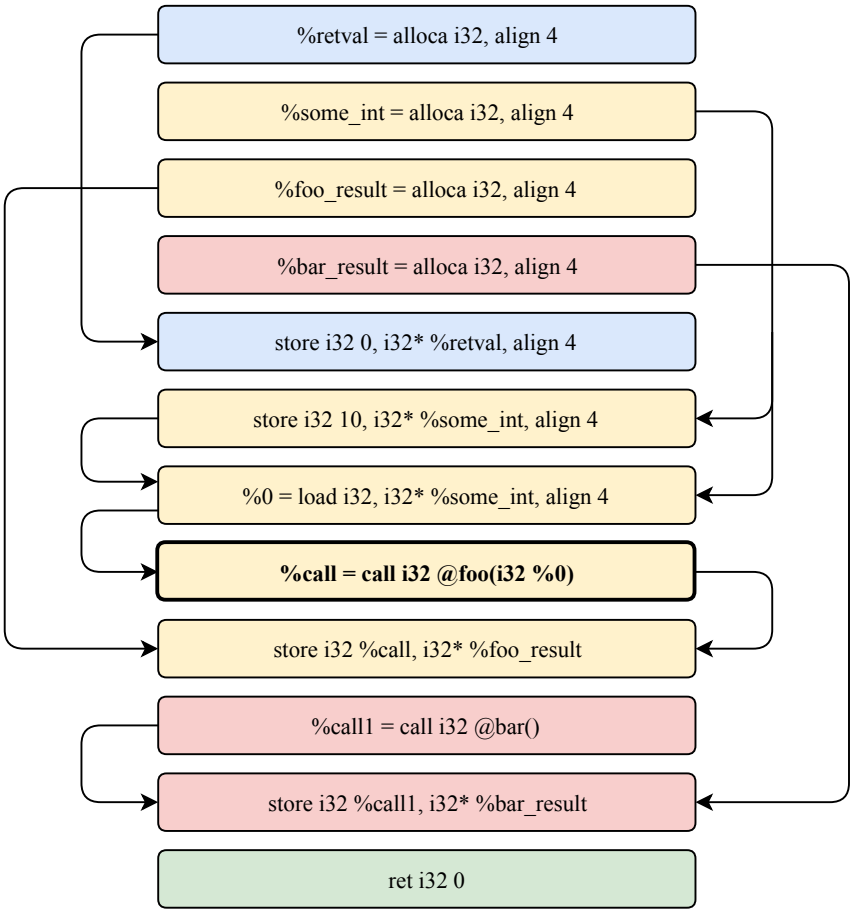

Figure 3.2: Connected components of the data dependencies graph for the `main` function instructions. Individual components are differentiated by the color.

Splitting instructions of each function into separated connected components is useful especially because we can answer the question if some particular instruction is in data dependency relationship with multiple other instructions (instructions that form a data dependency path, etc.).

3.5 Computing Call Graph

The next important part of our method is to compute feasible path from the source to the target that can be taken by the program execution. In order to simplify this, we propose to compute a call graph.

A **call graph**[5] is a control flow graph that represents relationship between program procedures[SSK09]. Having call graph $G = \{V, E\}$, the set of vertices V typically represents functions in the program and set of the edges E represents the transfer of control flow from one function to another (i.e. a function call).

In our context, we use a slightly different call graph definition that better suits our needs. Our call graph represents a relationship between individual connected components (computed in the section 3.4) and functions that are being called from these components (by one of their instructions). In other words, our call graph is a mapping from components to functions.

We construct edges in the call graph using the following algorithm:

computing call graph

```
0. Let FS = set of functions in the code
1. Let CS(f) = set of components inside function f
2. FOR EACH function F in the set FS:
      FOR EACH component C in the set CS(F):
         FOR EACH instruction I in the component C:
            IF instruction I is a call instruction to some function X:
                Add edge (C, X) to E.
```

Running the presented algorithm on the data dependence graph of the `main` function that we computed earlier (Figure 3.2) produces a call graph structure shown in Figure 3.3. The `main` function contains four distinct components. We can see from the computed call graph, that `foo` is being called from the *yellow* component, and `bar` is being called from the *red* component.

3.6 Finding Path from the Source to the Target

As we mentioned earlier, we use the call graph defined in the previous section for finding program path between the source and the target instructions.

*"A **path** is a simple graph whose vertices can be arranged in a linear sequence in such a way that two vertices are adjacent if they are consecutive in the sequence, and are nonadjacent otherwise."*[BM08] In our case, path is a linear sequence of vertices consists of components computed in section 3.4.

5. More technically, *call multigraph* [SSK09].

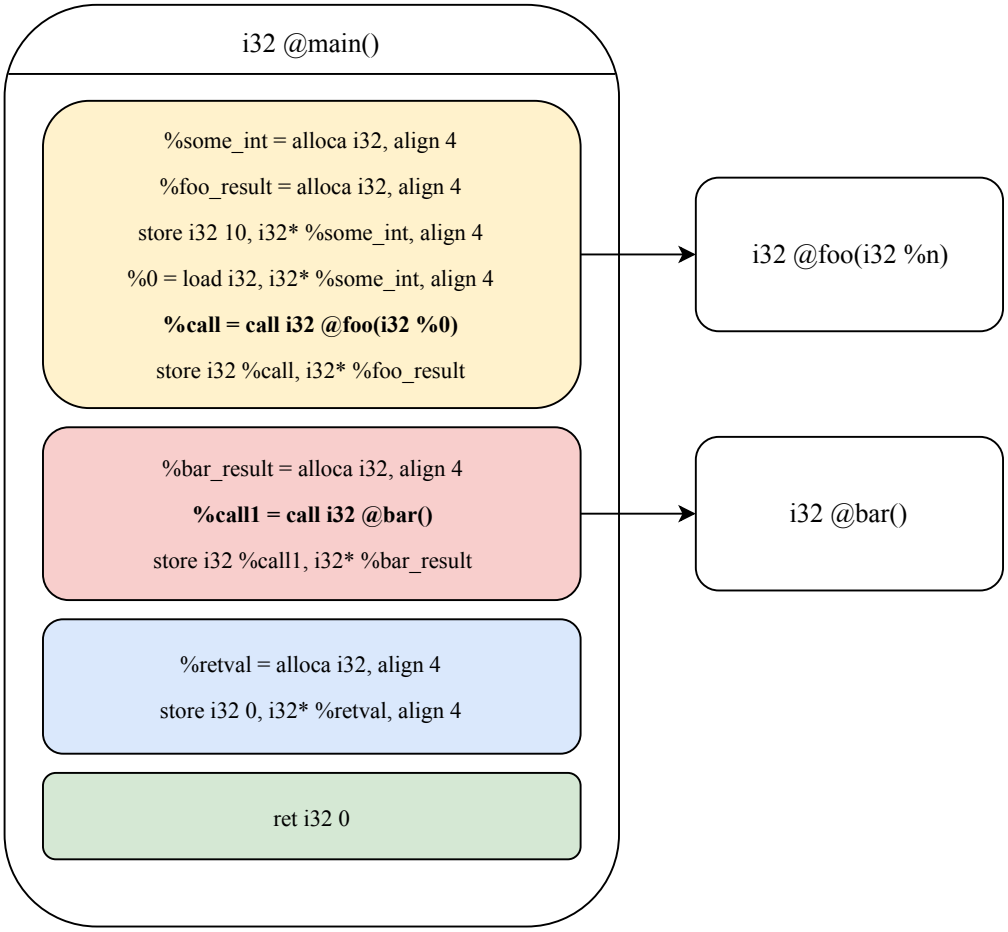

Figure 3.3: Call graph computed from the connected components as shown in the Figure 3.2.

Since the connected components contain related instructions, it is sufficient for us to compute a path in our call graph.

Potentially, there may exist an infinite number of such paths. From the optimization standpoint, it would be fitting to find all paths if possible and pick some path according to some optimization criteria (shortest path, path with smallest connected components, etc.). However, for our purposes, it will be sufficient to find any path, because our method does not try to optimize final code in respect to size, speed, etc. One path may potentially not be enough because of branching. In order to cover every branch, method would have to be extended.

To find a path, we traverse the call graph using the standard Breadth-first search algorithm.

Again, we illustrate the method at our running example in Figure 3.4. We can see that the path only contains two components, one from the `main` function and another from the function `bar`. These two components will be the core of the final, extracted program.

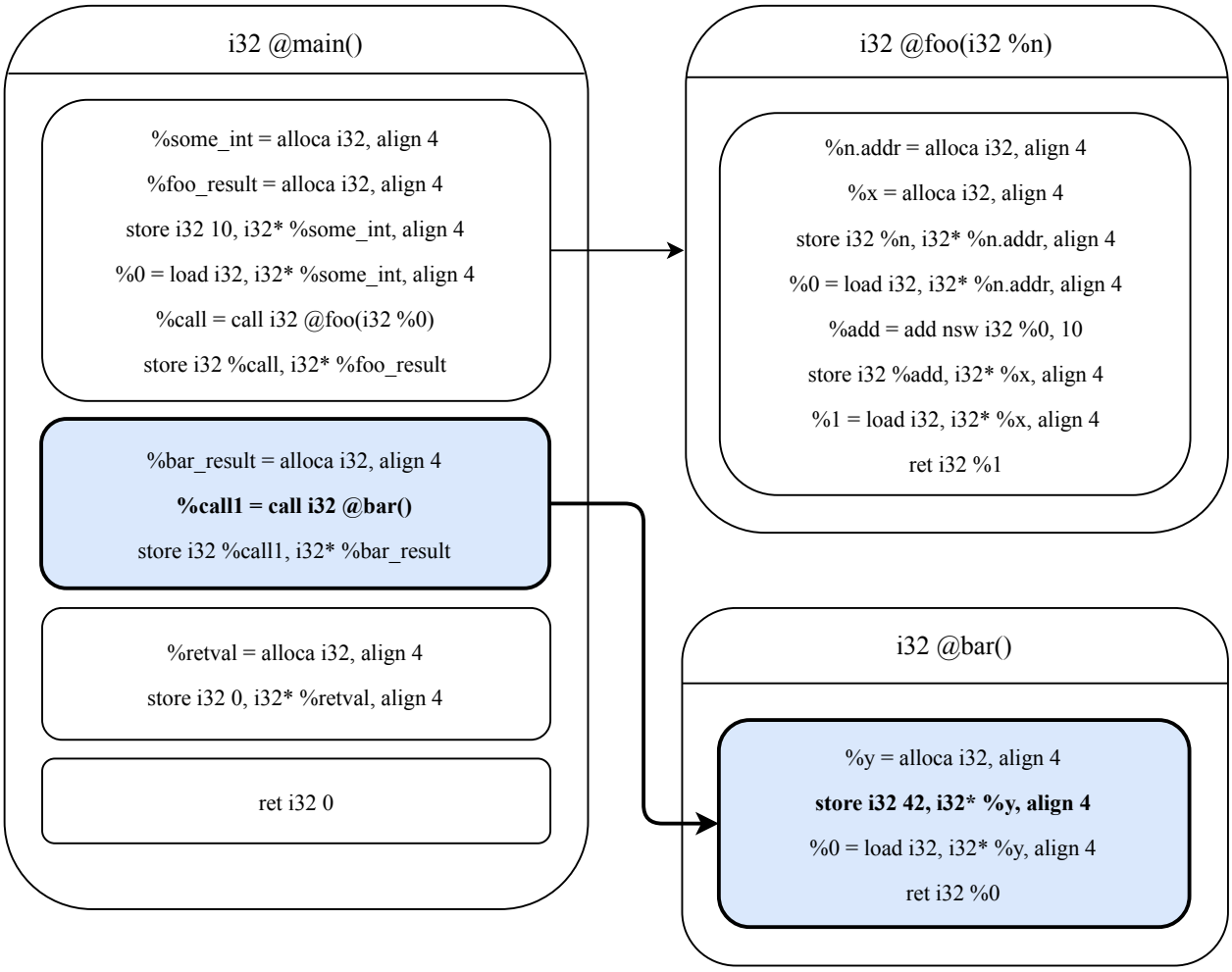

Figure 3.4: Path from source to target computed on the Figure 3.3. Components belonging to the path are *blue*.

3.7 Eliminating Dead Components and Functions

The final step of our method is extraction of the desired parts of code. We do this by eliminating so-called dead components and functions.

An element of the code is defined as **dead** if and only if its removal does not break reachability of target from source and integrity of the code.

In other words, we can safely remove dead elements from the code without being worried that the remaining code cannot be compiled and that the execution starting from source will not reach target.

In the previous section, we described how to find a path from source to target. Since this path should be preserved, each component that is a part of the path cannot be marked as dead. Therefore, components outside of the path have to be explored and the decision must be made whether they can be marked as dead or not.

We use the following basic algorithm for finding and eliminating dead components:

eliminating dead components

```
0. Let CS = set of all components in the code
1. Let PATH = set of components on the path from source to target
2. FOR EACH component C in set CS:
      IF component C is not in the set PATH:
         IF component C does not contain terminator:
             Mark component C as DEAD
3. Remove all components marked as DEAD
```

After applying the presented algorithm for eliminating dead components on the code from `example.s` we get the result that is shown in Figure 3.6. Components marked as dead are colored *red* because they are not a part of the path. The *yellow* component with the single instruction stands out. This component is not marked as dead and will not be removed because it contains a terminator instruction `ret i32 0`.[6] Finally, all components marked as dead are removed and we are left with the code that we presented in section 3.2 (`example_extracted.s`). Call graph of the `example_extracted.s` is visualized in Figure 3.5.

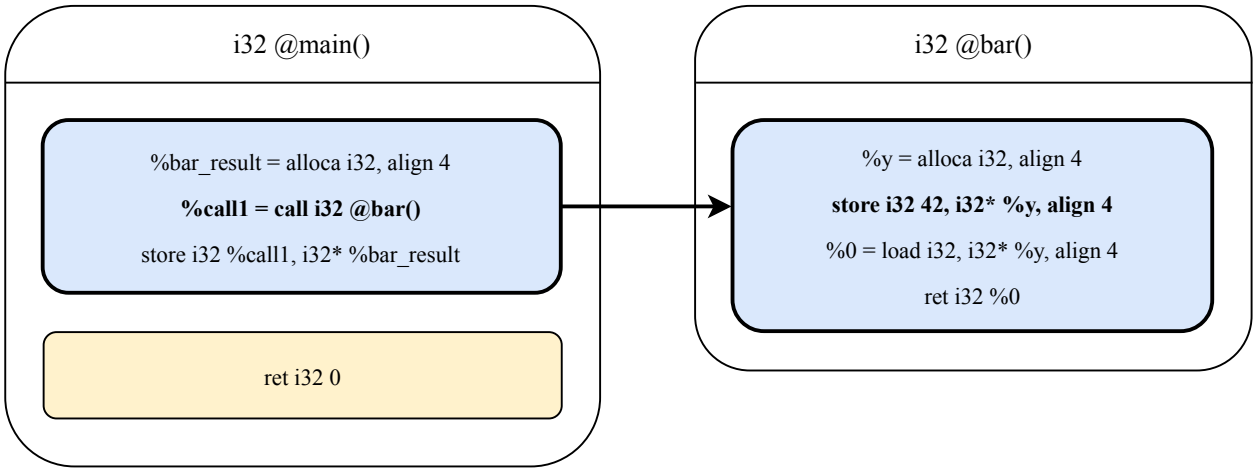

Figure 3.5: `example_extracted.s`: Final state after removing dead components and functions. `example.s`.

Unfortunately, the basic algorithm presented above only works correctly with non-complex inputs such as the one in `example.s`.

We present two extensions of the `example.s` program along with improvements of the basic algorithm that help to handle these extensions.

6. https://llvm.org/doxygen/classllvm_1_1TerminatorInst.html

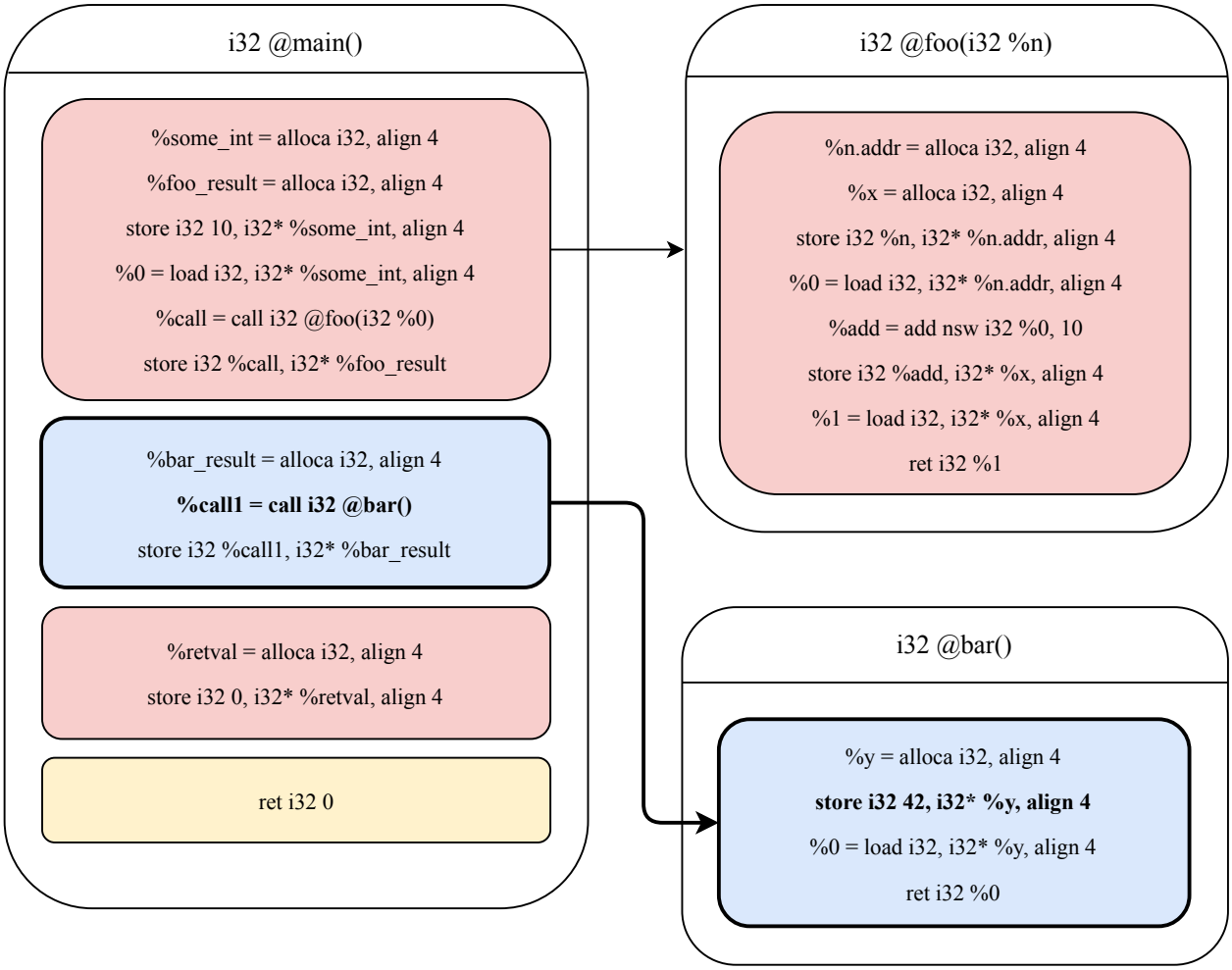

Figure 3.6: `example.s`: Components selected for removal are marked *red*. *Yellow* component is not marked for removal, because it contains a terminator instruction.

3.7.1 Path Depending on the External Function

The first problem is that there may exist a function that is not part of the path but that is called by one instruction on the path. In such case, when looking upon Figure 3.7, we see that function `bar` calls function `qux`, but `qux` is not part of the computed path (only components with the *blue* color are part of the path, that is one component from `main` and one from `foo`).

This is problematic for the basic algorithm that we introduced in the section 3.7. This basic algorithm would mark `qux` as dead and subsequently remove this function. However, this would break code integrity, because function `qux` has to be called in order for the execution to correctly proceed. Therefore, we propose modified algorithm for eliminating dead components that will take into the account the above described possibility.

eliminating dead components - mod1

```
0. Let CS = set of all components in the code
1. Let PATH = set of components on the path from source to target
2. Recursively find all called functions that originate from PATH
   using Breath-first search and add them to the PATH.
3. FOR EACH component C in set CS:
      IF component C is not in the set PATH:
         IF component C does not contain terminator:
             Mark component C as DEAD
4. Remove all components marked as DEAD
```

We modify the `example.c` by introducing a new function `int qux(void)`. Also, we change line 7 of the `example.c` from `int y = 42` to `int y = qux()`, the resulting code is shown in `example_mod1.c`:

example_mod1.c

```
1  int foo(int n) {
2      int x = n + 10;
3      return x;
4  }
5
6  int qux(void) {
7      return 42;
8  }
9
10 int bar(void) {
11     int y = qux();
12     return y;
13 }
14
```

```
15 int main(void) {
16     int some_int = 10;
17     int foo_result = foo(some_int);
18     int bar_result = bar();
19
20     return 0;
21 }
```

Target stays the same line as in the original example from section 3.2. This corresponds to line 11 in the `example_mod1.c`.

The LLVM IR of `example_mod1.c` is shown in `example_mod1.s` and its call graph with computed components and path can be seen in Figure 3.7.

By using the above described algorithm instead of the basic one from section 3.7, the path will contain the function `qux` and therefore, the function `qux` will not be marked as dead and removed. We can see the result of the algorithm in Figure 3.8 with the corresponding code in `example_mod1_extracted.s`. The function `qux` has not been removed which means that the integrity of the code was preserved and an executable could be successfully produced.

example_mod1.s

```
1  define i32 @foo(i32 %n) #0 {
2  entry:
3    %n.addr = alloca i32, align 4
4    %x = alloca i32, align 4
5    store i32 %n, i32* %n.addr, align 4
6    %0 = load i32, i32* %n.addr, align 4
7    %add = add nsw i32 %0, 10
8    store i32 %add, i32* %x, align 4
9    %1 = load i32, i32* %x, align 4
10   ret i32 %1
11 }
12
13 define i32 @qux() #0 {
14 entry:
15   ret i32 42
16 }
17
18 define i32 @bar() #0 {
19 entry:
20   %y = alloca i32, align 4
21   %call = call i32 @qux()
22   store i32 %call, i32* %y, align 4
23   %0 = load i32, i32* %y, align 4
```

```
  ret i32 %0
}

define i32 @main() #0 {
entry:
  %retval = alloca i32, align 4
  %some_int = alloca i32, align 4
  %foo_result = alloca i32, align 4
  %bar_result = alloca i32, align 4
  store i32 0, i32* %retval, align 4
  store i32 10, i32* %some_int, align 4
  %0 = load i32, i32* %some_int, align 4
  %call = call i32 @foo(i32 %0)
  store i32 %call, i32* %foo_result, align 4
  %call1 = call i32 @bar()
  store i32 %call1, i32* %bar_result, align 4
  ret i32 0
}
```

example_mod1_extracted.s

```
define i32 @qux() #0 {
entry:
  ret i32 42
}

define i32 @bar() #0 {
entry:
  %y = alloca i32, align 4
  %call = call i32 @qux()
  store i32 %call, i32* %y, align 4
  %0 = load i32, i32* %y, align 4
  ret i32 %0
}

define i32 @main() #0 {
entry:
  %bar_result = alloca i32, align 4
  %call1 = call i32 @bar()
  store i32 %call1, i32* %bar_result, align 4
  ret i32 0
}
```

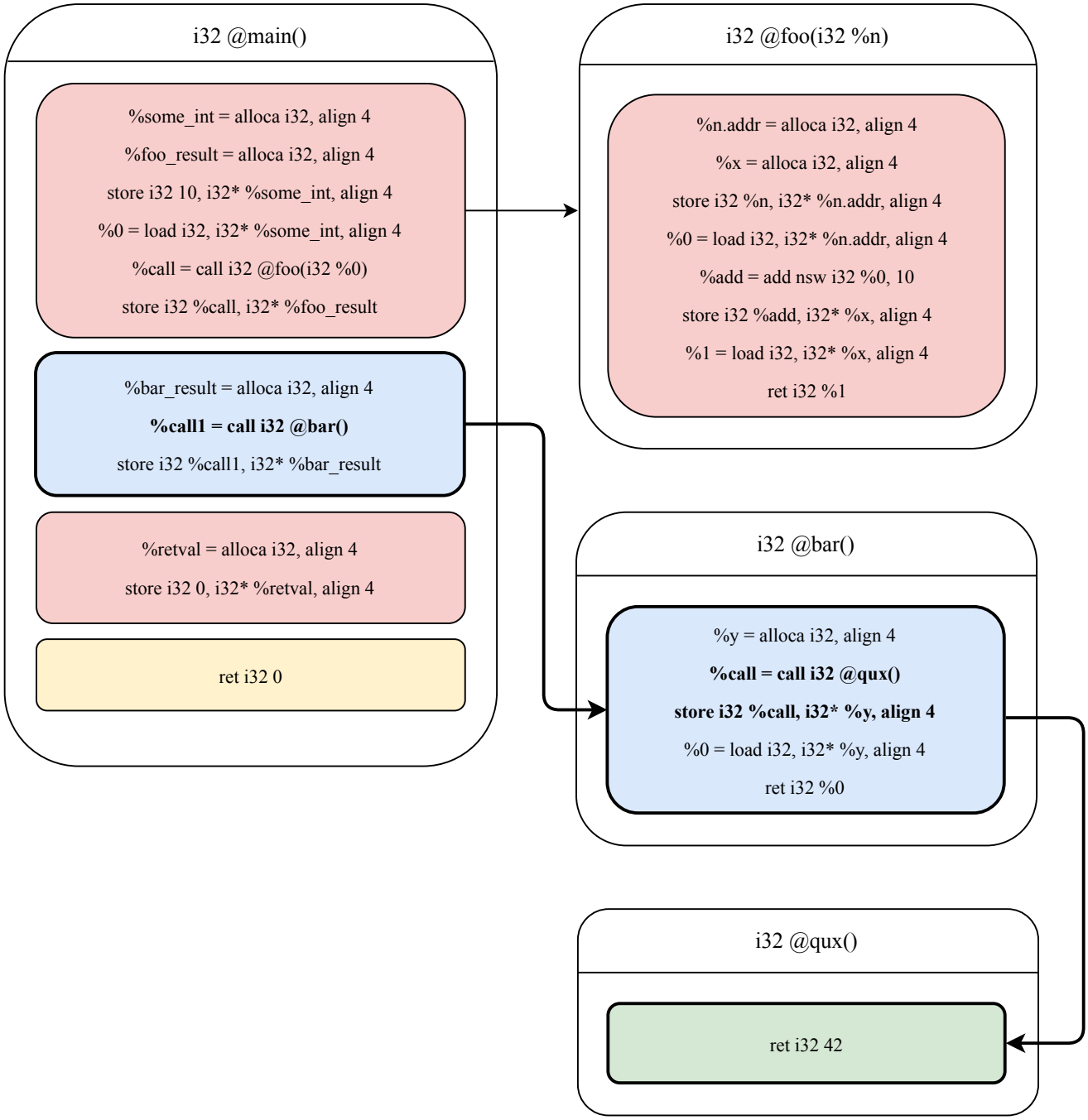

Figure 3.7: `example_mod1.s`: Components selected for removal are marked *red*. *Yellow* component is not marked for removal, because it contains terminator. *Green* component was discovered by the modified algorithm and has been added to the path and therefore will not be removed.

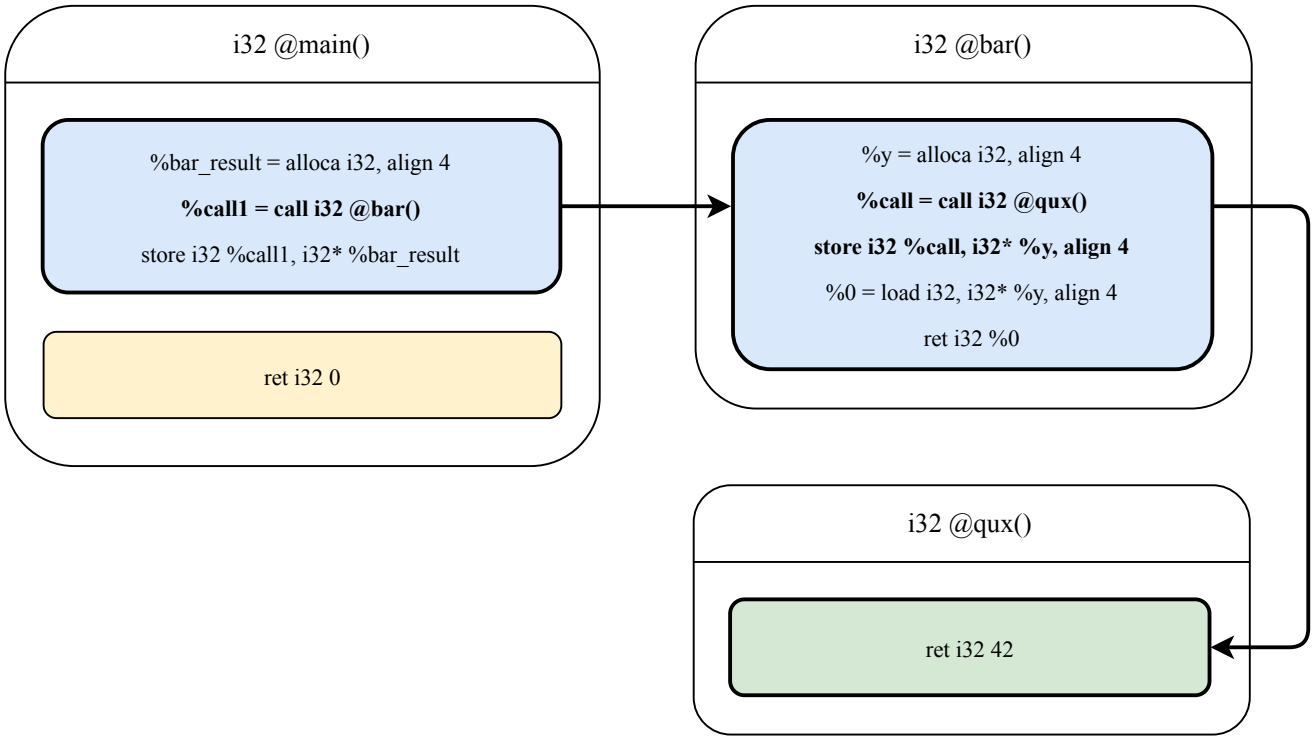

Figure 3.8: `example_mod1_extracted.s`: Final state after removing dead components and functions from `example_mod1.s`.

3.7.2 Path Depending on a Branching Instruction

Another problem comes with usage of branching instructions.

We now introduce branching into the main function of our examples shown in code `example_mod2.c`. The corresponding LLVM IR is shown in the `example_mod2.s` and the branch instruction occurs at line 42. Also, there appears two new basic blocks that this branch instruction refers to: `if.then` and `if.end` (lines 44 and 49, respectively).

If we take a closer look at the generated call graph for `example_mod2.s` (Figure 3.10), we can observe that the branching instruction `br i1 %cmp, label %if.then, label %if.end` is in the component that is not part of the path. This is problematic, because if we look at the `example_mod2.c`. we can clearly see, that in order to get to the target (line 11), branching needs to be executed and thus included in the path. We present the algorithm from the previous subsection (subsection 3.7.1) with the extension that handles the above described scenario:

```
eliminating dead components - mod2
0. Let CS = set of all components in the code
1. Let PATH = set of components on the path from source to target
2. FOR EACH component C in PATH:
       FOR EACH instruction I in C:
```

```
          IF instruction I is part of basic block handled by the branch
           ↪ instruction:
             FIND component with the branching instruction responsible
              ↪ for I and add it to the PATH
3. Recursively find all called functions that originate from PATH
   using Breath-first search and add them to the PATH.
4. FOR EACH component C in set CS:
      IF component C is not in the set PATH:
         IF component C does not contain terminator:
             Mark component C as DEAD
5. Remove all components marked as DEAD
```

As we can see in the `example_mod2.s`, if we examine instruction from the line 45, we can see that it belongs to the basic block `if.then`.

The call graph for the new program is shown in the Figure 3.10. We can observe that the instruction from line 45 lies on the path and at the same time belongs to the basic block `if.then`. This basic block is handled by the branch instruction from line 42. The modified algorithm finds a component that contains this branch instruction and adds it to the path. The call graph with the computed components and the path that the algorithm takes as the input is shown in Figure 3.10. The component in the main function marked as green contains a branching instruction that the algorithm identified as necessary and therefore is was added to the path.

The result of the algorithm is presented in `example_mod2_extracted.s` and visually in the form of a call graph Figure 3.9.

example_mod2.c

```c
int foo(int n) {
    int x = n + 10;
    return x;
}

int qux(void) {
    return 42;
}

int bar(void) {
    int y = qux();
    return y;
}

int main(void) {
    int some_int = 10;
    int foo_result = foo(some_int);
```

```
18      int n = 10;
19      if (n < 42) {
20          int bar_result = bar();
21      }
22      return 0;
23  }
```

example_mod2.s

```
1   define i32 @foo(i32 %n) #0 {
2   entry:
3     %n.addr = alloca i32, align 4
4     %x = alloca i32, align 4
5     store i32 %n, i32* %n.addr, align 4
6     %0 = load i32, i32* %n.addr, align 4
7     %add = add nsw i32 %0, 10
8     store i32 %add, i32* %x, align 4
9     %1 = load i32, i32* %x, align 4
10    ret i32 %1
11  }
12
13  define i32 @qux() #0 {
14  entry:
15    ret i32 42
16  }
17
18  define i32 @bar() #0 {
19  entry:
20    %y = alloca i32, align 4
21    %call = call i32 @qux()
22    store i32 %call, i32* %y, align 4
23    %0 = load i32, i32* %y, align 4
24    ret i32 %0
25  }
26
27  define i32 @main() #0 {
28  entry:
29    %retval = alloca i32, align 4
30    %some_int = alloca i32, align 4
31    %foo_result = alloca i32, align 4
32    %n = alloca i32, align 4
33    %bar_result = alloca i32, align 4
34    store i32 0, i32* %retval, align 4
35    store i32 10, i32* %some_int, align 4
36    %0 = load i32, i32* %some_int, align 4
```

```
  %call = call i32 @foo(i32 %0)
  store i32 %call, i32* %foo_result, align 4
  store i32 10, i32* %n, align 4
  %1 = load i32, i32* %n, align 4
  %cmp = icmp slt i32 %1, 42
  br i1 %cmp, label %if.then, label %if.end

if.then:                                          ; preds = %entry
  %call1 = call i32 @bar()
  store i32 %call1, i32* %bar_result, align 4
  br label %if.end

if.end:                                           ; preds = %if.then, %entry
  ret i32 0
}
```

example_mod2_extracted.s

```
define i32 @qux() #0 {
entry:
  ret i32 42
}

define i32 @bar() #0 {
entry:
  %y = alloca i32, align 4
  %call = call i32 @qux()
  store i32 %call, i32* %y, align 4
  %0 = load i32, i32* %y, align 4
  ret i32 %0
}

define i32 @main() #0 {
entry:
  %n = alloca i32, align 4
  %bar_result = alloca i32, align 4
  store i32 10, i32* %n, align 4
  %0 = load i32, i32* %n, align 4
  %cmp = icmp slt i32 %0, 42
  br i1 %cmp, label %if.then, label %if.end

if.then:                                          ; preds = %entry
  %call1 = call i32 @bar()
  store i32 %call1, i32* %bar_result, align 4
  br label %if.end
```

```
28
29 if.end:                                          ; preds = %if.then, %entry
30   ret i32 0
31 }
```

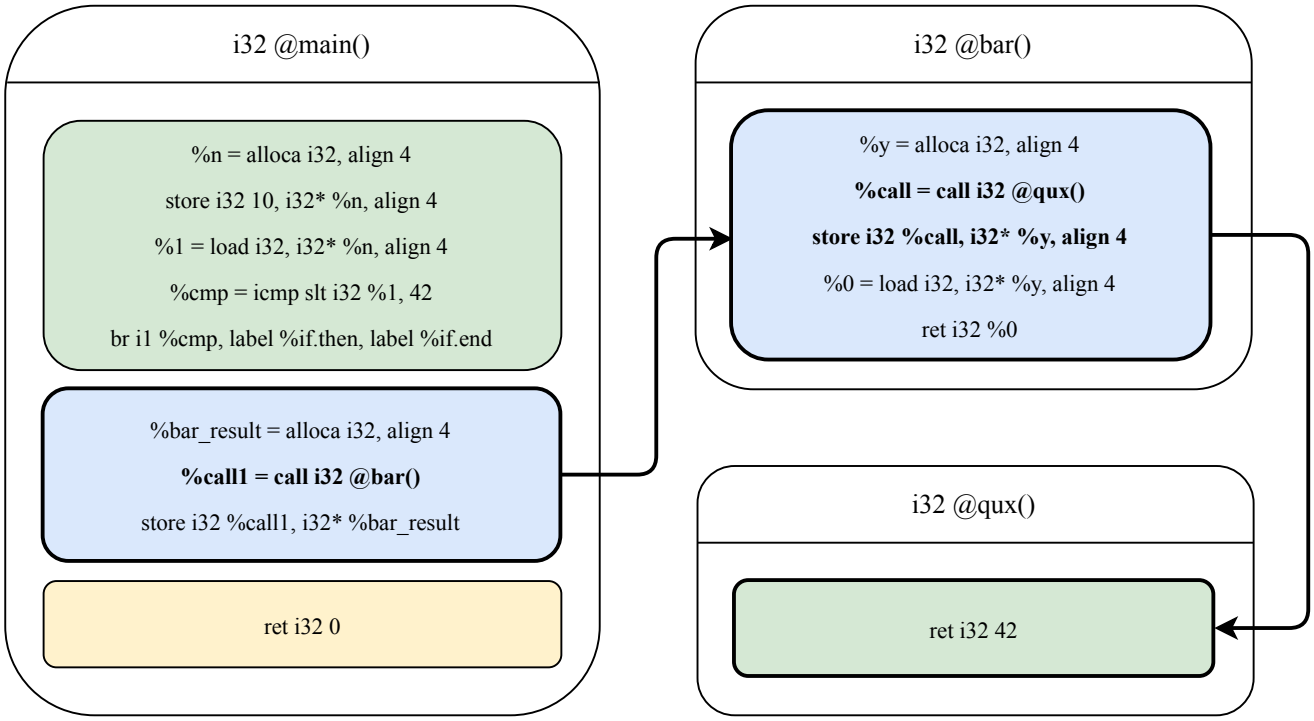

Figure 3.9: `example_mod2_extracted.s`: Final state after removing dead components and functions from `example_mod2.s`.

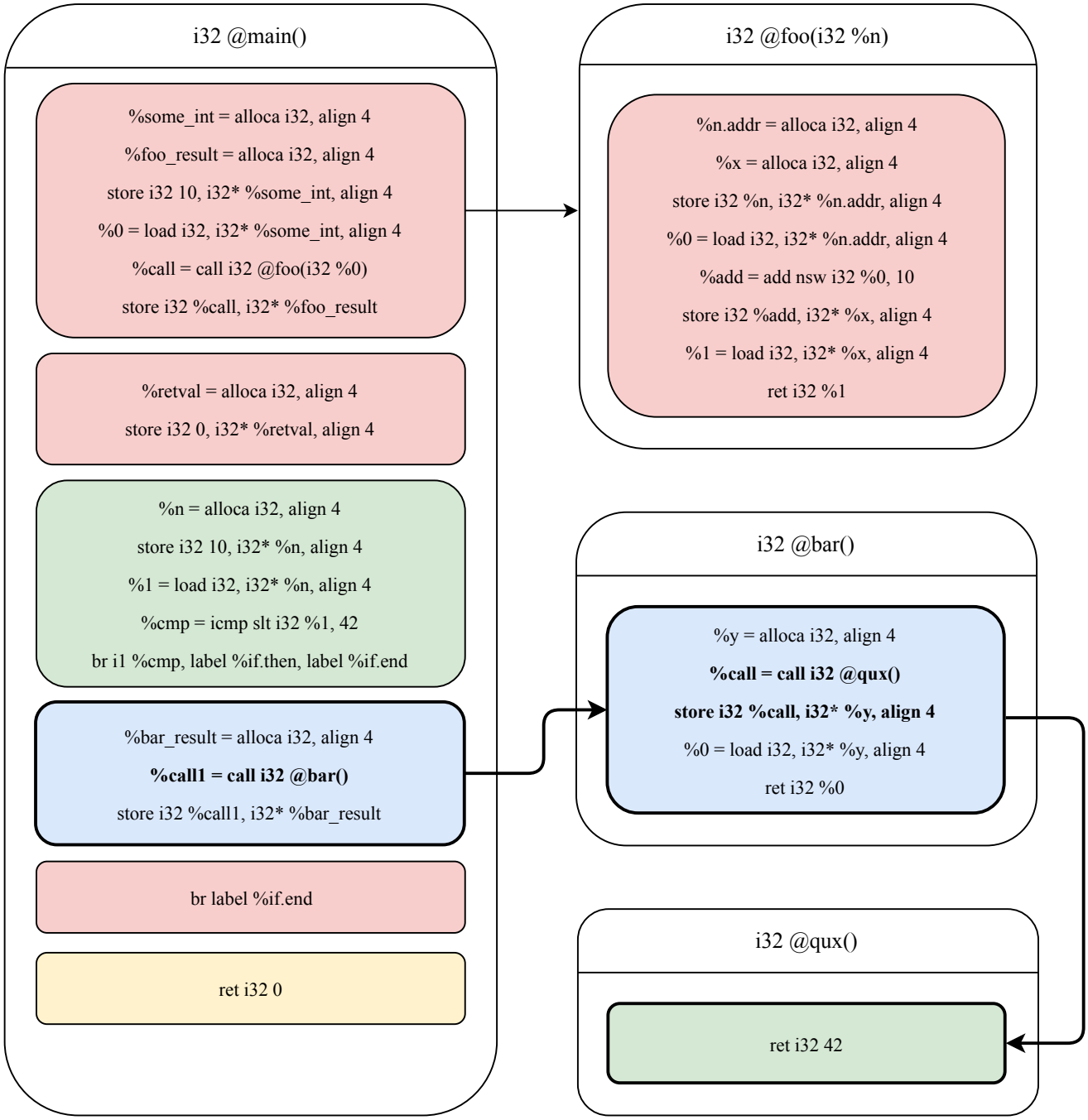

Figure 3.10: `example_mod2.s`: Components selected for removal are marked *red*. *Yellow* component is not marked for removal, because it a contains terminator instruction. *Green* components were discovered by the modified algorithm and were added to the path and therefore will not be removed.

4 Implementation

4.1 APEXPass

The method that we described in the chapter 3 is implemented as a LLVM pass called **APEXPass**. We have decided to implement the method as an pass because of the great capabilities of the LLVM infrastructure. Leveraging LLVM, APEXPass can be ran as an optimization step along with other LLVM optimizations or executed separately with LLMV optimizer tool opt:

```
opt -o apex.bc -load libAPEXPass.so -apex -file=example.c -line=7 <
↪  example.bc 2> build/apex.log
```

Using opt command presented above, APEXPass is ran via **apex** flag and defines two command line arguments: file and line. Argument **file** specifies the C source file where is the target located (in our case example.c) and argument **line** specifies number of the line in the file (in the example, line number 7).

Input to the opt is the example.bc, which is the bitcode representation of the example.c with included source-level debug information. The reason why we need debug information included in the example.bc is explained in the subsection 4.1.2.

We can get example.bc by compiling example.c using clang:

```
clang -g -c -emit-llvm example.c -o example.bc
```

Another necessary input to the opt is **libAPEXPass.so**. It is the pass itself compiled into the shared library.

APEXPass produces output **apex.bc**, which is the bitcode representation of the extracted part from the example.bc. This extracted program can be executed directly with the tool lli as follows:

```
lli apex.bc
```

APEXPass can be defined within the LLVM framework as a **ModulePass** because it derives from the ModulePass class.[1]

1. https://llvm.org/doxygen/classllvm_1_1ModulePass.html

There are other pass clases besides ModulePass (FunctionPass, LoopPass, RegionPass, BasicBlockPass, etc.), but we have decided to implement APEXPass as a ModulePass because it is the most general class of passes and thus provides the greatest flexibility.

There are some disadvantages that comes to implementing pass as an ModulePass. To name a few, function bodies are are referred by no particular order. Also, there are no possible optimizations to be done for ModulePass because optimizer does not have enough information about its behaviors. [LLV18f]

However, these disadvantages are not critical for our purposes and advantages of having whole program abstracted as a single unit outweigh the disadvantages.

4.1.1 Structure of the APEXPass

In order to work correctly, each ModulePass has to override runOnModule method with the following signature:

```
virtual bool runOnModule(Module &M) = 0;
```

This method is the main part of the pass and performs computation of the pass. All five steps that we described in the section 3.1 are going to be part of the `runOnModule` method. In this section, we are going to explain additional, implementation specific steps that needed to be added to the method.

We add three additional steps on top of the original five method steps described in the section 3.1 (step 1, 7 and 8). We also explain implementation specific details behind step 2 in the subsequent subsection.

The APEXPass structure therefore looks like this:

- `runOnModule()`:
  1. **Locate target instructions that map to the user input.**
  2. **Run dg.** Compute data dependencies between instructions.
  3. Find connected components in the computed data dependencies inside every function.
  4. Construct a call graph - a mapping between connected components and functions that are being called from these components.
  5. Find a path from the source to the target in the call graph.
  6. Eliminate dead components and functions that do not depend on the path.
  7. **Inject exit and extract function calls into the code.**
  8. **Strip debug symbols from the extracted code.**

4.1.2 Locating Target Instructions

Since APEXPass, or generally any LLVM pass, does not work directly with the C code but instead works with the LLVM IR, we need a procedure for mapping C source code to the IR.

As we mentioned in the section 3.2, we know precise position of the target in the example.c, but we do knot know exectly what IR instructions are equivalent to the target.

Taking example from the section 3.2, we have input program example.c and target in the function bar at the line 7.

The target in the example.c is the following line of code: `int y = 42;` which is represented by the following IR instruction in the example.s:

```
store i32 42, i32* %y, align 4
```

The way we achieve mapping between original C source code and IR is by using source-level debug information that is introduced by the compiler (usually with the -g flag). The example.s with debug symbols is shown in the Appendix C. Once the example.s is compiled with debug information, we can iterate from within the APEXPass over each instruction to find out its parent file name and line number using the following code:[LLV18g]

```
for (const auto &function : module) {
    for (const auto &basic_block : function) {
        for (const auto &instruction : basic_block) {
            if (DILocation *Loc = instruction->getDebugLoc()) {
                unsigned Line = Loc->getLine();
                StringRef File = Loc->getFilename();
            }
        }
    }
}
```

Once the Line and File match with the user input, we have found our target IR instruction.

Depending on how complex the target line of code is, there may be more than one IR instructions associated with it. For example, taking the target line number 7 from the example.c:

```
int y = 42;
```

IR instruction that maps to it is the following:

```
store i32 42, i32* %y, align 4
```

However, when we take as a target line number 2 from the example.c:

```
int x = n + 10;
```

Then, the mapping will be to the three following IR instructions:

```
%0 = load i32, i32* %n.addr, align 4
%add = add nsw i32 %0, 10
store i32 %add, i32* %x, align 4
```

`int x = n + 10;` is much more complex line of code than `int y = 42;` and thus, is represented by more IR instructions.

4.1.3 dg - Computing Data Dependencies

As we have mentioned in the section 3.3, we have not implemented data dependency computation in the APEXPass itself. Instead, we use open-source tool by Marek Chalupa called **dg**[2] to compute data dependencies and get the results that we described in the section 3.3.

Dg is a library which implements dependence graphs for programs. It contains a set of generic templates that can be specialized to user's needs. As a part of dg, you can find pointer analyses, reaching definitions analysis and a static slicer for LLVM.[CHA18]

Since dg is extensive project that covers more than we need, we use just some parts of it, especially LLVMDependenceGraph. This structure provides data dependencies which APEXPass requires.

4.1.4 Injecting Exit and Extract Functions

After having found the target instruction, we need to ensure that the execution of the extracted program stops right after the target has been reached. We do not want to execute program after it reached the target, because it is contradictory to what our method should do. We also want to extract value of the target and provide this information to the user.

In order to accomplish this, we need to inject the following three additional instructions right after the target:

- Load instruction X that takes value of the target and stores it into the separate register.
- Call instruction to extract function with X as an function argument.
- Call instruction to exit function which will terminate execution.

Taking example_extracted.s from the section 3.2 (with target `store i32 42, i32* %y, align 4`) and injecting exit and extract functions after the target gives the final result presented in the example_extracted_final.s code:

example_extracted_final.s

2. https://github.com/mchalupa/dg

```
1  define i32 @bar() #0 {
2  entry:
3    %y = alloca i32, align 4
4    store i32 42, i32* %y, align 4
5    %_apex_extract_int_arg = load i32, i32* %y
6    call void @_apex_extract_int(i32 %_apex_extract_int_arg)
7    call void @_apex_exit(i32 0)
8    %0 = load i32, i32* %y, align 4
9    ret i32 %0
10 }
11
12 define i32 @main() #0 {
13 entry:
14   %bar_result = alloca i32, align 4
15   %call1 = call i32 @bar()
16   store i32 %call1, i32* %bar_result, align 4
17   ret i32 0
18 }
```

As we can see on the lines 5,6 and 7, there are three new instructions injected right after the target. Instructions at lines 6 and 7 are call instructions to the extract and exit functions. Because these instructions call external functions, their definitions has to be linked to the program. This is done by the apex.py and is discussed in the section 4.2.

Exit function `_apex_exit` and extract function `_apex_extract_int` are implemented in the library `apexlib.c`:

apexlib.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  void _apex_exit(int exit_code) {
5    exit(exit_code);
6  }
7
8  void _apex_extract_int(int i) {
9    printf("%d", i);
10 }
```

Extraction is limited by the type of extracted value. By looking at the `example extracted final.` we can observe that the type of `%y` is `i32` (integer). We have to call extraction function that accepts argument of this type. Current implementation supports only extraction of values with the integer type (more specifically `i32`).

4.1.5 Stripping Debug Symbols

The last step that APEXPass performs is stripping debug information from the final code. The reason why we do this step is to be sure that the final extracted program is in the consistent state. APEXPass works with the code that contains debug symbols. Problem is that the three new injected instructions that were presented in the subsection 4.1.4 are without debug symbols. Having parts of the code with debug symbols and other parts without creates inconsistent state. Program in this inconsistent state would not be possible to compile into bitcode without issues.

There are two approaches that could resolve this issue:

1. Add debug information to each new instruction that we insert into the code.
2. Strip all debug information from the code.

We have picked the second option, because it is sufficient to ensure code consistency and because of the ease of the implementation. LLVM provides API[3] for stripping debug information from function, therefore we can use the following code to strip all debug information from the entire program:

```
for (auto &Function : Module) {
    llvm::stripDebugInfo(Function);
}
```

4.2 APEX

We have integrated APEXPass into the open-source tool called APEX, which is freely accessible in the repository: https://github.com/examon/APEX. APEX serves as an user-friendly wrapper around APEXPass while providing an easy way to run APEXPass without leaving user to worry about the internal structure.

4.2.1 apex.py

There are multiple steps that need to be executed in order to properly run APEXPass. Therefore, we have implemented script apex.py that simplifies this process to the absolute minimum.

Running APEXPass via launcher apex.py does require only Python interpreter. The following example demonstrates how to run APEXPass with the input example.bc, with target located in the file example.c on the line 2.

3. https://llvm.org/doxygen/DebugInfo_8cpp_source.html#l00314

```
python apex.py example.bc example.c 2 --export=true
```

apex.py launcher has three mandatory command line arguments and one optional. First argument is input code in the LLVM bitcode format. Second argument is the target file and third is target line of code in the file. Optional argument is boolean flag and when set to true, apex.py will export call graphs of the input and also the extracted program.

Upon finishing execution, apex.py will create executable called **extracted**. This executable contains extracted part of the program and can be run just like any unix executable:

```
./extracted
```

Besides producing expected executable, apex.py also creates build directory with the following important files:

- apex.bc: Extracted program in the LLVM bitcode format that can be run separately via tool lli.
- apex.out: Output of the apex.bc.
- apex.log: Log produced by the APEXPass.

4.2.2 Linking apexlib and Input

The important step in the apex.py execution is linking apexlib with the user input. Linking has to be done because apexlib defines exit and extract functions and these functions need to be injected after the target (subsection 4.1.4).

Linking itself is performed in the following three setps:

1. Compile apexlib.c into the LLVM bitcode apexlib.bc.
   `clang -g -c -emit-llvm apexlib.c -o apexlib.bc`
2. Link apexlib.bc with the user input (e.g. example.bc) and produce linked.ll IR code.
   `llvm-link apexlib.bc example.bc -S -o=linked.ll`
3. Run assember on the linked.ll and produce bitcode linked.bc.
   `llvm-as build/linked.ll -o build/linked.bc`

These three steps will essentially take user input example.bc, link to it library with the exit and extract functions apexlib.bc and produce output linked.bc that will now serve as an input to the APEXPass instead of the original input example.bc.

linked.ll is available for comparison with example.s in the Appendix D.

5 Experiments

We present four experiments in this chapter. The aim of the experiments is to test APEX on various inputs and determine if the implementation can handle extracting parts from the basic programs. We perform extractions on various targets and evaluate the results.

To demonstrate APEX capabilities, we perform tests on one artificially created program and three well know UNIX utilities.

5.1 example_mod2.c

The first experiment is conducted on the code example_mod2.c, which we introduced in subsection 3.7.2. We picked this simple code in order to demonstrate the whole process of testing APEX capabilities.

example_mod2.c

```
1  int foo(int n) {
2      int x = n + 10;
3      return x;
4  }
5
6  int qux(void) {
7      return 42;
8  }
9
10 int bar(void) {
11     int y = qux();
12     return y;
13 }
14
15 int main(void) {
16     int some_int = 10;
17     int foo_result = foo(some_int);
18     int n = 10;
19     if (n < 42) {
20         int bar_result = bar();
21     }
22     return 0;
23 }
```

First, we compile example_mod2.bc into the LLVM bitcode with debug information:

```
clang -c -g -emit-llvm example_mod2.c -o example_mod2.bc
```

We run -dot-callgraph pass via opt tool and dot with the following commands to generate call graph image from the example_mod2.bc (Figure 5.1):

```
opt -dot-callgraph example_mod2.bc && dot callgraph.dot -Tsvg -O
```

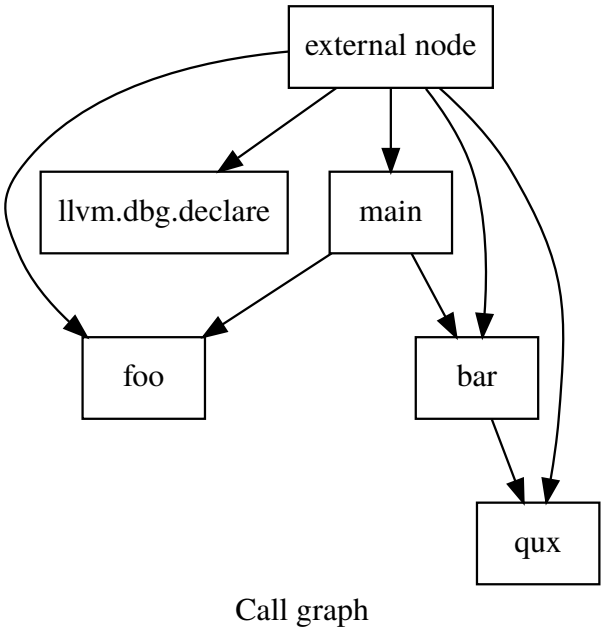

Figure 5.1: `example_mod2.c` call graph before linking and running APEXPass.

Figure 5.1 serves as as reference for the subsequent stages of this experiment. Having compiled input with debug symbols successfully, we fulfilled requirements for APEX input and can therefore run apex.py to start extraction.

extraction with target line number 16

```
> python apex.py example_mod2.bc example_mod2.c 16
> ./extracted
10
```

APEX generated binary `extracted` with respect to the target at the line number 16. When executing extracted binary, we get value of the `some_int` via injected function `_apex_extract_int` (which is 10). Execution subsequently stops because `_apex_exit` is called. APEX also generated build/apex.bc file, which is the extracted program in the LLVM bitcode format. We can generate the call graph using the same method as we did with the example_mod2.bc and get Figure 5.2.

We can see, that the generated call graph includes functions from apexlib, namely `_apex_exit` and `_apex_extract_int`.

When comparing Figure 5.1 and Figure 5.2 we observe, that functions bar and qux are deleted. APEXPass determined that they do not need to be part of the final extracted executable and therefore dropped them from the code.

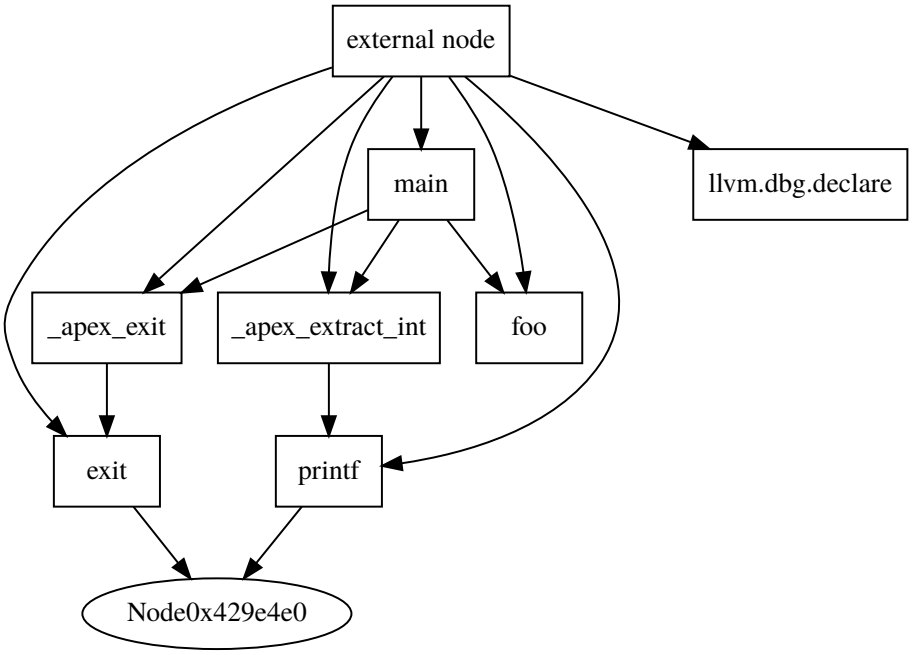

Figure 5.2: `example_mod2.c`: call graph after extraction with target at the line 16.

5.2 `yes.c`

The following experiment will be conducted on the classic UNIX utility called yes. yes repeatedly outputs a line with specified string or 'y' until killed.

We use source code from the open-source implementation of yes from the following repository: https://github.com/mubaris/yes

Full C source code that implements yes is shown in the yes.c listing:

yes.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  #define max(a,b) \
6      ({ __typeof__ (a) _a = (a); \
7         __typeof__ (b) _b = (b); \
8         _a > _b ? _a : _b; });
9
10 int main(int argc, char* argv[]) {
11     char* output;
12     int output_size;
13     int output_len;
14
```

```
15      if (argc > 1) {
16          // Create buffer.
17          int needed_size = 2 + strlen(argv[1]);
18          for (int i = 2; i < argc; i++) {
19              needed_size += 1 + strlen(argv[i]);
20          }
21
22          output = (char*) malloc(needed_size);
23
24          // Append to buffer.
25          strcat(output, argv[1]);
26          for (int i = 2; i < argc; i++) {
27              strcat(output, " ");
28              strcat(output, argv[i]);
29          }
30          strcat(output, "\n");
31          output_len = strlen(output);
32      } else {
33          output = "y\n";
34          output_len = 2;
35      }
36
37      // Flood.
38      for(;;)
39          fwrite(output, 1, output_len, stdout);
40  }
```

After compilation yes.c into the bitcode yes.bc, we pick `int needed_size` from line 17 as a target.

extraction with target line number 17

```
> python apex.py yes.bc yes.c 17
> ./extracted foo
5
> ./extracted foobar
8
> ./extracted
y
y
y
...
```

Extracted program correctly outputs the value of the `int needed_size` when we provide some command line argument. It also correctly executes the branch else when we do

not provide any command line argument.

Next, we pick line 31 as a target, which corresponds to the line `output_len = strlen(output);`. Running apex.py, we get the following output when executing extracted binary:

```
extraction with target line number 31
> python apex.py yes.bc yes.c 31
> ./extracted foo
4
> ./extracted foobar
7
> ./extracted
y
y
y
...
```

Finally, lets test the else branch of the yes.c by picking line 34 as a target: (`output_len = 2;`):

```
extraction with target line number 34
> python apex.py yes.bc yes.c 34
> ./extracted foo
foo
foo
foo
...
> ./extracted
2
```

From the performed experimentation, we can conclude that APEX can handle inputs with the complexity of the yes.c source code.

5.3 domainname.c

Second UNIX utility that we use for testing is domainname, which is used for showing or setting the system's NIS/YP domain name.

We use the following open-source implementation of domainname provided by the OpenBSD community: https://raw.githubusercontent.com/openbsd/src/master/bin/domainname/domainname.c

domainname.c

```c
#include <err.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <limits.h>

char *__progname;
void usage(void);

int
main(int argc, char *argv[])
{
    int ch;
    char domainname[HOST_NAME_MAX+1];

    while ((ch = getopt(argc, argv, "")) != -1)
        switch (ch) {
        default:
            usage();
        }
    argc -= optind;
    argv += optind;

    if (argc > 1)
        usage();

    if (*argv) {
        if (setdomainname(*argv, strlen(*argv)))
            err(1, "setdomainname");
    } else {
        if (getdomainname(domainname, sizeof(domainname)))
            err(1, "getdomainname");
        (void)printf("%s\n", domainname);
    }
    return(0);
}


void usage(void)
{
    (void)fprintf(stderr, "usage: %s [name-of-domain]\n", __progname);
    exit(1);
}
```

Picking target `argc -= optind;` from line 22, we get the following results:

extraction with target line number 22

```
python apex.py domainname.bc domainname.c 22
> ./extracted
0
> ./extracted a b
2
> ./extracted a b c d
4
```

Interestingly, since we targeted `argc` variable, it looks like the extracted executable counts number of supplied command line arguments.

5.4 sleep.c

Last UNIX utility that we test is sleep. This utility produces delay for a specified amount of time.

We use the following open-source implementation of sleep provided by the OpenBSD community: https://raw.githubusercontent.com/openbsd/src/master/bin/sleep/sleep.c

domainname.c

```
1  #include <ctype.h>
2  #include <signal.h>
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <time.h>
6  #include <unistd.h>
7  #include <err.h>
8
9  extern char *__progname;
10 void usage(void);
11 void alarmh(int);
12
13 int
14 main(int argc, char *argv[])
15 {
16     int ch;
```

```
17      time_t secs = 0, t;
18      char *cp;
19      int nsecs = 0;
20      struct timespec rqtp;
21      int i;
22
23      signal(SIGALRM, alarmh);
24
25      while ((ch = getopt(argc, argv, "")) != -1)
26          switch(ch) {
27          default:
28              usage();
29      }
30      argc -= optind;
31      argv += optind;
32
33      if (argc != 1)
34          usage();
35
36      cp = *argv;
37      while ((*cp != '\0') && (*cp != '.')) {
38          if (!isdigit((unsigned char)*cp))
39              errx(1, "seconds is invalid: %s", *argv);
40          t = (secs * 10) + (*cp++ - '0');
41          if (t / 10 != secs)         /* oflow */
42              errx(1, "seconds is too large: %s", *argv);
43          secs = t;
44      }
45
46      /* Handle fractions of a second */
47      if (*cp == '.') {
48          cp++;
49          for (i = 100000000; i > 0; i /= 10) {
50              if (*cp == '\0')
51                  break;
52              if (!isdigit((unsigned char)*cp))
53                  errx(1, "seconds is invalid: %s", *argv);
54              nsecs += (*cp++ - '0') * i;
55          }
56
57          while (*cp != '\0') {
58              if (!isdigit((unsigned char)*cp++))
59                  errx(1, "seconds is invalid: %s", *argv);
60          }
61      }
```

```
62
63      while (secs > 0 || nsecs > 0) {
64          if (secs > 100000000) {
65              rqtp.tv_sec = 100000000;
66              rqtp.tv_nsec = 0;
67          } else {
68              rqtp.tv_sec = secs;
69              rqtp.tv_nsec = nsecs;
70          }
71          if (nanosleep(&rqtp, NULL))
72              err(1, NULL);
73          secs -= rqtp.tv_sec;
74          nsecs -= rqtp.tv_nsec;
75      }
76      return (0);
77  }
78
79  void
80  usage(void)
81  {
82      (void)fprintf(stderr, "usage: %s seconds\n", __progname);
83      exit(1);
84  }
85
86  void
87  alarmh(int signo)
88  {
89      _exit(0);
90  }
```

By picking target `nsecs += (*cp++ - '0') * i;` at line 54, we get the following results:

extraction with target line number 54

```
python apex.py sleep.bc sleep.c 54
> ./extracted foo
extracted: seconds is invalid: foo
> ./extracted 3
...
(sleeps 3 seconds)
...
> ./extracted 3.14
100000000
> ./extracted
Segmentation fault (core dumped)
```

Running extracted with string or integer argument produces the expected behavior (error and sleeping). When running extracted executable with command line argument encompassing decimal point, the execution hits our target at the line 54. Value of the `nsecs` is extracted via `_apex_extract_int` function and printed as expected.

The interesting thing happens when we run extracted binary without any command line arguments. If we look into sleep.c, we can see that in the case where there are no command line arguments, usage function should be called. However, because we have picked target at the line 54, APEXPass determined that it is safe to remove function usage from the code (which is the case when we run binary with command line arguments).

If we compare call graphs Figure 5.3 and Figure 5.4, we can see that the function usage has been removed and is no longer part of the extracted binary.

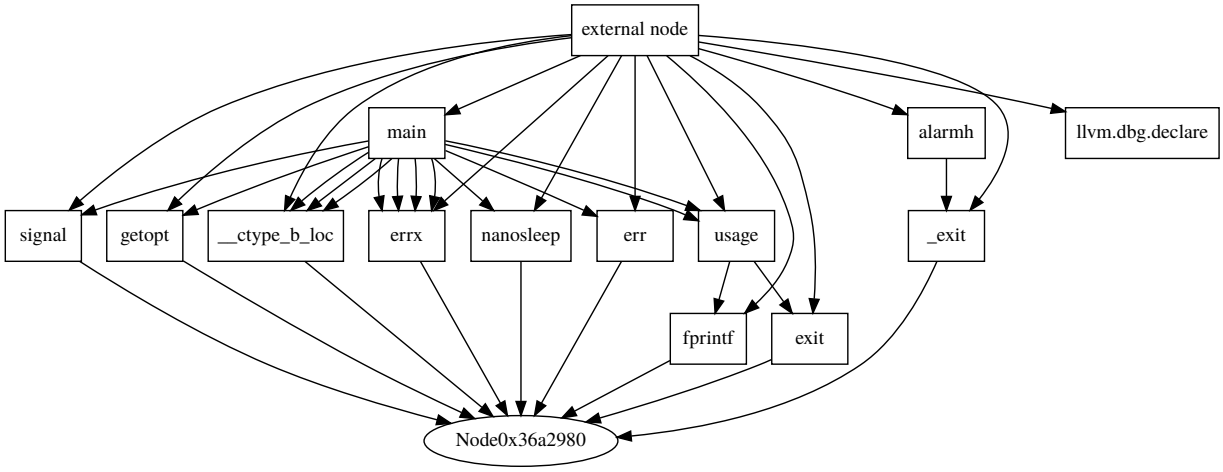

Figure 5.3: `sleep.c` call graph before linking and running APEXPass.

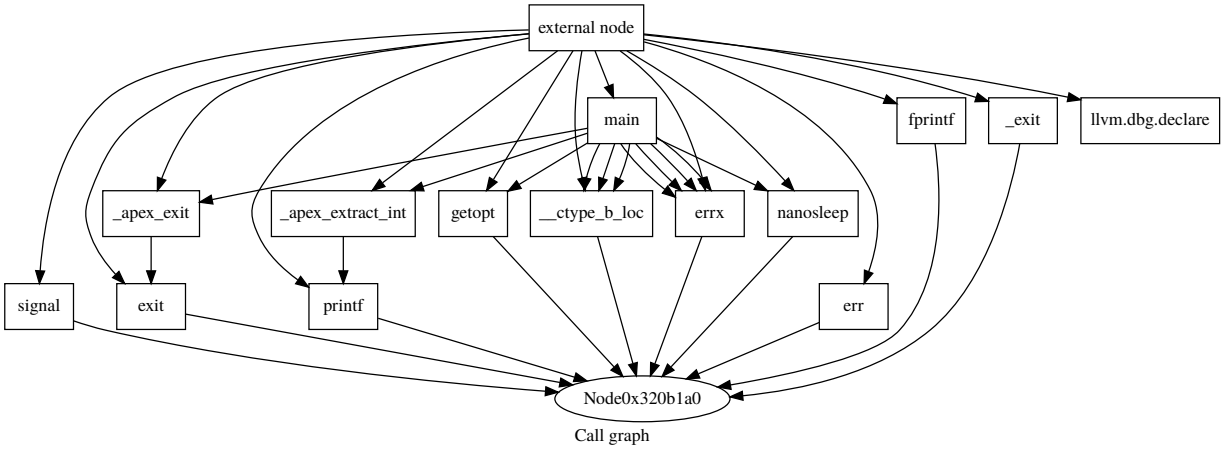

Figure 5.4: `sleep.c` call graph after extraction with target at the line 54.

6 Conclusions

In this thesis, we presented the method of extracting parts of programs into separate binaries. We explained steps and algorithms that the method performs to achieve the desired extraction.

We developed a LLVM transform pass (APEXPass) implementing the presented method by leveraging the LLVM infrastructure. APEXPass was integrated into the open-source tool APEX, which provides the user-friendly experience.

Lastly, we have tested APEX on one artificially created program and three classic UNIX tools. Experiments showed, that the method implemented in the APEX is sufficient for extracting parts of programs from simple inputs that are comparable in complexity to small UNIX tools.

6.1 Further Research and Development

As we mentioned in the subsection 4.1.4, extraction is currently limited by the type of the extracted value. Possible improvement of the extraction procedure would be to implement extraction functions for every IR type and thus extending apexlib.c. Implementing algorithm able to determine what type of extraction function is needed would be crucial in order to support all value types for extraction.

Path finding procedure that we described in section 3.6 could be improved by finding multiple paths (not only one) and picking the best fitting one for the extraction. The best fitting path could be determined by some optimization criteria (e.g. shortest path, path without fewest external dependencies, etc.). Finding multiple paths would also help with the branching coverage.

Another very useful improvement would be to extend coverage of the dead code elimination procedure explained in section 3.7. We have already presented two extensions to this procedure (subsection 3.7.1 and subsection 3.7.2). Designing even more extensions would allow the method to cover more complex IR code than now and thus would allow extraction from more complicated C programs.

Bibliography

[BM08] J. Bondy and U. Murty, *Graph theory*, 1st. Springer Publishing Company, Incorporated, 2008, ISBN: 1846289696.

[CHA18] M. CHALUPA. (2018). Dependence graph for programs. Generic implementation of dependence graphs with instantiation for LLVM that contains a static slicer for LLVM bitcode, [Online]. Available: https://github.com/mchalupa/dg (visited on 12/03/2018).

[CHA25] ——, "Slicing of llvm bitcode [online]", Master's thesis, Masaryk University, Faculty of Informatics, Brno, 2016 [cit. 2018-11-25]. [Online]. Available: Available%20from%20WWW%20%3Chttps://is.muni.cz/th/vik1f/%3E.

[Lat18] C. Lattner. (2018). LLVM - The Architecture of Open Source Applications, [Online]. Available: http://aosabook.org/en/llvm.html (visited on 11/03/2018).

[LLV18a] LLVM. (2018). Clang: a C language family frontend for LLVM, [Online]. Available: https://clang.llvm.org/ (visited on 12/03/2018).

[LLV18b] ——, (2018). Introduction to the Clang AST, [Online]. Available: https://clang.llvm.org/docs/IntroductionToTheClangAST.html (visited on 12/03/2018).

[LLV18c] ——, (2018). LLVM Language Reference Manual, [Online]. Available: https://llvm.org/docs/LangRef.html (visited on 12/03/2018).

[LLV18d] ——, (2018). llvm::BasicBlock Class Reference, [Online]. Available: https://llvm.org/doxygen/classllvm_1_1BasicBlock.html (visited on 12/03/2018).

[LLV18e] ——, (2018). LLVM's Analysis and Transform Passes, [Online]. Available: https://llvm.org/docs/Passes.html (visited on 11/09/2018).

[LLV18f] ——, (2018). LLVM's Analysis and Transform Passes, [Online]. Available: https://llvm.org/docs/WritingAnLLVMPass.html#the-modulepass-class (visited on 11/09/2018).

[LLV18g] ——, (2018). Source Level Debugging with LLVM, [Online]. Available: https://llvm.org/docs/SourceLevelDebugging.html#c-c-source-file-information (visited on 12/03/2018).

[LLV18h] ——, (2018). The LLVM Compiler Infrastructure, [Online]. Available: https://llvm.org/ (visited on 11/03/2018).

[Ras10] F. Rastello, *Ssa-based compiler design*. New York London: Springer, 2010, ISBN: 978-1-4419-6201-0.

[SSK09] A. Sanyal, B. Sathe, and U. Khedker, *Data flow analysis: Theory and practice*. CRC Press, 2009.

A Archive structure

Archive contains APEX source code. Upstream version of the APEX can be found in the repository: https://github.com/examon/APEX

Please refer to the `README.md` for instructions about building and running APEX.

Example source codes used in this thesis are included in the directory `examples`.

B example.s

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

; Function Attrs: noinline nounwind optnone uwtable
define i32 @foo(i32 %n) #0 {
entry:
  %n.addr = alloca i32, align 4
  %x = alloca i32, align 4
  store i32 %n, i32* %n.addr, align 4
  %0 = load i32, i32* %n.addr, align 4
  %add = add nsw i32 %0, 10
  store i32 %add, i32* %x, align 4
  %1 = load i32, i32* %x, align 4
  ret i32 %1
}

; Function Attrs: noinline nounwind optnone uwtable
define i32 @bar() #0 {
entry:
  %y = alloca i32, align 4
  store i32 42, i32* %y, align 4
  %0 = load i32, i32* %y, align 4
  ret i32 %0
}

; Function Attrs: noinline nounwind optnone uwtable
define i32 @main() #0 {
entry:
  %retval = alloca i32, align 4
  %some_int = alloca i32, align 4
  %foo_result = alloca i32, align 4
  %bar_result = alloca i32, align 4
  store i32 0, i32* %retval, align 4
  store i32 10, i32* %some_int, align 4
  %0 = load i32, i32* %some_int, align 4
  %call = call i32 @foo(i32 %0)
  store i32 %call, i32* %foo_result, align 4
  %call1 = call i32 @bar()
  store i32 %call1, i32* %bar_result, align 4
```

```
42    ret i32 0
43  }
44
45  attributes #0 = { noinline nounwind optnone 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" }
46
47  !llvm.module.flags = !{!0}
48  !llvm.ident = !{!1}
49
50  !0 = !{i32 1, !"wchar_size", i32 4}
51  !1 = !{!"clang version 5.0.1 (tags/RELEASE_500/final)"}
```

C example.s - with debug symbols

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

; Function Attrs: noinline nounwind optnone uwtable
define i32 @foo(i32 %n) #0 !dbg !7 {
entry:
  %n.addr = alloca i32, align 4
  %x = alloca i32, align 4
  store i32 %n, i32* %n.addr, align 4
  call void @llvm.dbg.declare(metadata i32* %n.addr, metadata !11,
   ↪  metadata !12), !dbg !13
  call void @llvm.dbg.declare(metadata i32* %x, metadata !14, metadata
   ↪  !12), !dbg !15
  %0 = load i32, i32* %n.addr, align 4, !dbg !16
  %add = add nsw i32 %0, 10, !dbg !17
  store i32 %add, i32* %x, align 4, !dbg !15
  %1 = load i32, i32* %x, align 4, !dbg !18
  ret i32 %1, !dbg !19
}

; Function Attrs: nounwind readnone speculatable
declare void @llvm.dbg.declare(metadata, metadata, metadata) #1

; Function Attrs: noinline nounwind optnone uwtable
define i32 @bar() #0 !dbg !20 {
entry:
  %y = alloca i32, align 4
  call void @llvm.dbg.declare(metadata i32* %y, metadata !23, metadata
   ↪  !12), !dbg !24
  store i32 42, i32* %y, align 4, !dbg !24
  %0 = load i32, i32* %y, align 4, !dbg !25
  ret i32 %0, !dbg !26
}

; Function Attrs: noinline nounwind optnone uwtable
define i32 @main() #0 !dbg !27 {
entry:
  %retval = alloca i32, align 4
  %some_int = alloca i32, align 4
```

```
39    %foo_result = alloca i32, align 4
40    %bar_result = alloca i32, align 4
41    store i32 0, i32* %retval, align 4
42    call void @llvm.dbg.declare(metadata i32* %some_int, metadata !28,
      ↪  metadata !12), !dbg !29
43    store i32 10, i32* %some_int, align 4, !dbg !29
44    call void @llvm.dbg.declare(metadata i32* %foo_result, metadata !30,
      ↪  metadata !12), !dbg !31
45    %0 = load i32, i32* %some_int, align 4, !dbg !32
46    %call = call i32 @foo(i32 %0), !dbg !33
47    store i32 %call, i32* %foo_result, align 4, !dbg !31
48    call void @llvm.dbg.declare(metadata i32* %bar_result, metadata !34,
      ↪  metadata !12), !dbg !35
49    %call1 = call i32 @bar(), !dbg !36
50    store i32 %call1, i32* %bar_result, align 4, !dbg !35
51    ret i32 0, !dbg !37
52  }
53
54  attributes #0 = { noinline nounwind optnone 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" }
55  attributes #1 = { nounwind readnone speculatable }
56
57  !llvm.dbg.cu = !{!0}
58  !llvm.module.flags = !{!3, !4, !5}
59  !llvm.ident = !{!6}
60
61  !0 = distinct !DICompileUnit(language: DW_LANG_C99, file: !1, producer:
      ↪  "clang version 5.0.1 (tags/RELEASE_500/final)", isOptimized: false,
      ↪  runtimeVersion: 0, emissionKind: FullDebug, enums: !2)
62  !1 = !DIFile(filename: "example.c", directory:
      ↪  "/mnt/Documents/work/university/muni/msc/thesis/APEX/examples/example")
63  !2 = !{}
64  !3 = !{i32 2, !"Dwarf Version", i32 4}
65  !4 = !{i32 2, !"Debug Info Version", i32 3}
66  !5 = !{i32 1, !"wchar_size", i32 4}
67  !6 = !{!"clang version 5.0.1 (tags/RELEASE_500/final)"}
```

```
68 !7 = distinct !DISubprogram(name: "foo", scope: !1, file: !1, line: 1,
   ↪ type: !8, isLocal: false, isDefinition: true, scopeLine: 1, flags:
   ↪ DIFlagPrototyped, isOptimized: false, unit: !0, variables: !2)
69 !8 = !DISubroutineType(types: !9)
70 !9 = !{!10, !10}
71 !10 = !DIBasicType(name: "int", size: 32, encoding: DW_ATE_signed)
72 !11 = !DILocalVariable(name: "n", arg: 1, scope: !7, file: !1, line: 1,
   ↪ type: !10)
73 !12 = !DIExpression()
74 !13 = !DILocation(line: 1, column: 13, scope: !7)
75 !14 = !DILocalVariable(name: "x", scope: !7, file: !1, line: 2, type: !10)
76 !15 = !DILocation(line: 2, column: 9, scope: !7)
77 !16 = !DILocation(line: 2, column: 13, scope: !7)
78 !17 = !DILocation(line: 2, column: 15, scope: !7)
79 !18 = !DILocation(line: 3, column: 12, scope: !7)
80 !19 = !DILocation(line: 3, column: 5, scope: !7)
81 !20 = distinct !DISubprogram(name: "bar", scope: !1, file: !1, line: 6,
   ↪ type: !21, isLocal: false, isDefinition: true, scopeLine: 6, flags:
   ↪ DIFlagPrototyped, isOptimized: false, unit: !0, variables: !2)
82 !21 = !DISubroutineType(types: !22)
83 !22 = !{!10}
84 !23 = !DILocalVariable(name: "y", scope: !20, file: !1, line: 7, type:
   ↪ !10)
85 !24 = !DILocation(line: 7, column: 9, scope: !20)
86 !25 = !DILocation(line: 8, column: 12, scope: !20)
87 !26 = !DILocation(line: 8, column: 5, scope: !20)
88 !27 = distinct !DISubprogram(name: "main", scope: !1, file: !1, line: 11,
   ↪ type: !21, isLocal: false, isDefinition: true, scopeLine: 11, flags:
   ↪ DIFlagPrototyped, isOptimized: false, unit: !0, variables: !2)
89 !28 = !DILocalVariable(name: "some_int", scope: !27, file: !1, line: 12,
   ↪ type: !10)
90 !29 = !DILocation(line: 12, column: 9, scope: !27)
91 !30 = !DILocalVariable(name: "foo_result", scope: !27, file: !1, line: 13,
   ↪ type: !10)
92 !31 = !DILocation(line: 13, column: 9, scope: !27)
93 !32 = !DILocation(line: 13, column: 26, scope: !27)
94 !33 = !DILocation(line: 13, column: 22, scope: !27)
95 !34 = !DILocalVariable(name: "bar_result", scope: !27, file: !1, line: 14,
   ↪ type: !10)
96 !35 = !DILocation(line: 14, column: 9, scope: !27)
97 !36 = !DILocation(line: 14, column: 22, scope: !27)
98 !37 = !DILocation(line: 16, column: 5, scope: !27)
```

D linked.ll

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

@.str = private unnamed_addr constant [3 x i8] c"%d\00", align 1

; Function Attrs: noinline nounwind optnone uwtable
define void @_apex_exit(i32 %exit_code) #0 !dbg !9 {
entry:
  %exit_code.addr = alloca i32, align 4
  store i32 %exit_code, i32* %exit_code.addr, align 4
  call void @llvm.dbg.declare(metadata i32* %exit_code.addr, metadata !13,
   ↪  metadata !14), !dbg !15
  %0 = load i32, i32* %exit_code.addr, align 4, !dbg !16
  call void @exit(i32 %0) #4, !dbg !17
  unreachable, !dbg !17

return:                                           ; No predecessors!
  ret void, !dbg !18
}

; Function Attrs: nounwind readnone speculatable
declare void @llvm.dbg.declare(metadata, metadata, metadata) #1

; Function Attrs: noreturn nounwind
declare void @exit(i32) #2

; Function Attrs: noinline nounwind optnone uwtable
define void @_apex_extract_int(i32 %i) #0 !dbg !19 {
entry:
  %i.addr = alloca i32, align 4
  store i32 %i, i32* %i.addr, align 4
  call void @llvm.dbg.declare(metadata i32* %i.addr, metadata !20,
   ↪  metadata !14), !dbg !21
  %0 = load i32, i32* %i.addr, align 4, !dbg !22
  %call = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([3 x
   ↪  i8], [3 x i8]* @.str, i32 0, i32 0), i32 %0), !dbg !23
  ret void, !dbg !24
}

```

```
39 declare i32 @printf(i8*, ...) #3
40
41 ; Function Attrs: noinline nounwind optnone uwtable
42 define i32 @foo(i32 %n) #0 !dbg !25 {
43 entry:
44   %n.addr = alloca i32, align 4
45   %x = alloca i32, align 4
46   store i32 %n, i32* %n.addr, align 4
47   call void @llvm.dbg.declare(metadata i32* %n.addr, metadata !28,
     ↪ metadata !14), !dbg !29
48   call void @llvm.dbg.declare(metadata i32* %x, metadata !30, metadata
     ↪ !14), !dbg !31
49   %0 = load i32, i32* %n.addr, align 4, !dbg !32
50   %add = add nsw i32 %0, 10, !dbg !33
51   store i32 %add, i32* %x, align 4, !dbg !31
52   %1 = load i32, i32* %x, align 4, !dbg !34
53   ret i32 %1, !dbg !35
54 }
55
56 ; Function Attrs: noinline nounwind optnone uwtable
57 define i32 @bar() #0 !dbg !36 {
58 entry:
59   %y = alloca i32, align 4
60   call void @llvm.dbg.declare(metadata i32* %y, metadata !39, metadata
     ↪ !14), !dbg !40
61   store i32 42, i32* %y, align 4, !dbg !40
62   %0 = load i32, i32* %y, align 4, !dbg !41
63   ret i32 %0, !dbg !42
64 }
65
66 ; Function Attrs: noinline nounwind optnone uwtable
67 define i32 @main() #0 !dbg !43 {
68 entry:
69   %retval = alloca i32, align 4
70   %some_int = alloca i32, align 4
71   %foo_result = alloca i32, align 4
72   %bar_result = alloca i32, align 4
73   store i32 0, i32* %retval, align 4
74   call void @llvm.dbg.declare(metadata i32* %some_int, metadata !44,
     ↪ metadata !14), !dbg !45
75   store i32 10, i32* %some_int, align 4, !dbg !45
76   call void @llvm.dbg.declare(metadata i32* %foo_result, metadata !46,
     ↪ metadata !14), !dbg !47
77   %0 = load i32, i32* %some_int, align 4, !dbg !48
78   %call = call i32 @foo(i32 %0), !dbg !49
```

```
79    store i32 %call, i32* %foo_result, align 4, !dbg !47
80    call void @llvm.dbg.declare(metadata i32* %bar_result, metadata !50,
      ↪  metadata !14), !dbg !51
81    %call1 = call i32 @bar(), !dbg !52
82    store i32 %call1, i32* %bar_result, align 4, !dbg !51
83    ret i32 0, !dbg !53
84  }
85
86  attributes #0 = { noinline nounwind optnone 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" }
87  attributes #1 = { nounwind readnone speculatable }
88  attributes #2 = { noreturn nounwind
      ↪  "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" }
89  attributes #3 = { "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" }
90  attributes #4 = { noreturn nounwind }
91
92  !llvm.dbg.cu = !{!0, !3}
93  !llvm.ident = !{!5, !5}
94  !llvm.module.flags = !{!6, !7, !8}
95
96  !0 = distinct !DICompileUnit(language: DW_LANG_C99, file: !1, producer:
      ↪  "clang version 5.0.1 (tags/RELEASE_500/final)", isOptimized: false,
      ↪  runtimeVersion: 0, emissionKind: FullDebug, enums: !2)
```

```
97  !1 = !DIFile(filename: "src/apex/apexlib.c", directory:
    ↪  "/mnt/Documents/work/university/muni/msc/thesis/APEX")
98  !2 = !{}
99  !3 = distinct !DICompileUnit(language: DW_LANG_C99, file: !4, producer:
    ↪  "clang version 5.0.1 (tags/RELEASE_500/final)", isOptimized: false,
    ↪  runtimeVersion: 0, emissionKind: FullDebug, enums: !2)
100 !4 = !DIFile(filename: "example.c", directory:
    ↪  "/mnt/Documents/work/university/muni/msc/thesis/APEX/examples/example")
101 !5 = !{!"clang version 5.0.1 (tags/RELEASE_500/final)"}
102 !6 = !{i32 2, !"Dwarf Version", i32 4}
103 !7 = !{i32 2, !"Debug Info Version", i32 3}
104 !8 = !{i32 1, !"wchar_size", i32 4}
105 !9 = distinct !DISubprogram(name: "_apex_exit", scope: !1, file: !1, line:
    ↪  6, type: !10, isLocal: false, isDefinition: true, scopeLine: 6, flags:
    ↪  DIFlagPrototyped, isOptimized: false, unit: !0, variables: !2)
106 !10 = !DISubroutineType(types: !11)
107 !11 = !{null, !12}
108 !12 = !DIBasicType(name: "int", size: 32, encoding: DW_ATE_signed)
109 !13 = !DILocalVariable(name: "exit_code", arg: 1, scope: !9, file: !1,
    ↪  line: 6, type: !12)
110 !14 = !DIExpression()
111 !15 = !DILocation(line: 6, column: 21, scope: !9)
112 !16 = !DILocation(line: 7, column: 8, scope: !9)
113 !17 = !DILocation(line: 7, column: 3, scope: !9)
114 !18 = !DILocation(line: 8, column: 1, scope: !9)
115 !19 = distinct !DISubprogram(name: "_apex_extract_int", scope: !1, file:
    ↪  !1, line: 11, type: !10, isLocal: false, isDefinition: true,
    ↪  scopeLine: 11, flags: DIFlagPrototyped, isOptimized: false, unit: !0,
    ↪  variables: !2)
116 !20 = !DILocalVariable(name: "i", arg: 1, scope: !19, file: !1, line: 11,
    ↪  type: !12)
117 !21 = !DILocation(line: 11, column: 28, scope: !19)
118 !22 = !DILocation(line: 12, column: 16, scope: !19)
119 !23 = !DILocation(line: 12, column: 3, scope: !19)
120 !24 = !DILocation(line: 13, column: 1, scope: !19)
121 !25 = distinct !DISubprogram(name: "foo", scope: !4, file: !4, line: 1,
    ↪  type: !26, isLocal: false, isDefinition: true, scopeLine: 1, flags:
    ↪  DIFlagPrototyped, isOptimized: false, unit: !3, variables: !2)
122 !26 = !DISubroutineType(types: !27)
123 !27 = !{!12, !12}
124 !28 = !DILocalVariable(name: "n", arg: 1, scope: !25, file: !4, line: 1,
    ↪  type: !12)
125 !29 = !DILocation(line: 1, column: 13, scope: !25)
126 !30 = !DILocalVariable(name: "x", scope: !25, file: !4, line: 2, type:
    ↪  !12)
```

```
127 !31 = !DILocation(line: 2, column: 9, scope: !25)
128 !32 = !DILocation(line: 2, column: 13, scope: !25)
129 !33 = !DILocation(line: 2, column: 15, scope: !25)
130 !34 = !DILocation(line: 3, column: 12, scope: !25)
131 !35 = !DILocation(line: 3, column: 5, scope: !25)
132 !36 = distinct !DISubprogram(name: "bar", scope: !4, file: !4, line: 6,
    ↪ type: !37, isLocal: false, isDefinition: true, scopeLine: 6, flags:
    ↪ DIFlagPrototyped, isOptimized: false, unit: !3, variables: !2)
133 !37 = !DISubroutineType(types: !38)
134 !38 = !{!12}
135 !39 = !DILocalVariable(name: "y", scope: !36, file: !4, line: 7, type:
    ↪ !12)
136 !40 = !DILocation(line: 7, column: 9, scope: !36)
137 !41 = !DILocation(line: 8, column: 12, scope: !36)
138 !42 = !DILocation(line: 8, column: 5, scope: !36)
139 !43 = distinct !DISubprogram(name: "main", scope: !4, file: !4, line: 11,
    ↪ type: !37, isLocal: false, isDefinition: true, scopeLine: 11, flags:
    ↪ DIFlagPrototyped, isOptimized: false, unit: !3, variables: !2)
140 !44 = !DILocalVariable(name: "some_int", scope: !43, file: !4, line: 12,
    ↪ type: !12)
141 !45 = !DILocation(line: 12, column: 9, scope: !43)
142 !46 = !DILocalVariable(name: "foo_result", scope: !43, file: !4, line: 13,
    ↪ type: !12)
143 !47 = !DILocation(line: 13, column: 9, scope: !43)
144 !48 = !DILocation(line: 13, column: 26, scope: !43)
145 !49 = !DILocation(line: 13, column: 22, scope: !43)
146 !50 = !DILocalVariable(name: "bar_result", scope: !43, file: !4, line: 14,
    ↪ type: !12)
147 !51 = !DILocation(line: 14, column: 9, scope: !43)
148 !52 = !DILocation(line: 14, column: 22, scope: !43)
149 !53 = !DILocation(line: 16, column: 5, scope: !43)
```