Masarykova univerzita
Fakulta informatiky

Extracting Parts of Programs into Separate Binaries

Master's thesis

Tomáš Mészaroš

Brno, 2018

MASARYKOVA UNIVERZITA
Fakulta informatiky

ZADÁNÍ DIPLOMOVÉ PRÁCE

Student: Bc. Tomáš Mészaroš

Program: Aplikovaná informatika

Obor: Aplikovaná informatika

Specializace: Bez specializace

Garant oboru: prof. RNDr. Jiří Barnat, Ph.D. (AP)

Vedoucí práce: Mgr. Marek Grác, Ph.D.

Katedra: Katedra strojového učení a zpracování dat

Název práce: Extracting Parts of Programs into Separate Binaries

Název práce anglicky: Extracting Parts of Programs into Separate Binaries

Zadání:

- Get acquainted with means of the compilation of C programs using the LLVM compiler infrastructure - clang, LLVM Internal Representation, AST, LLVM optimizations.
- Propose a solution to statically transplant a subset of a C program. This subset should be extracted from the original program and synthesized as an independent binary.
- Design and implement the proposed solution in a tool having an appropriate form (a standalone application or an LLVM plugin).
- Test the implemented tool on at least 2 real-world open-source C programs.

Zadání bylo schváleno prostřednictvím IS MU.

Prohlášení autora školního díla

Jméno, příjmení a UČO studenta: Tomáš Mészaroš 422336

Beru na vědomí, že *Masarykova univerzita* může na základě zákona (§ 35 odst. 3 a 4 autorského zákona č. 121/2000 Sb.) užít mou kvalifikační práci nebo jiné mé školní dílo, které jsem jako autor vytvořil ke splnění svých studijních povinností vůči této vysoké škole, a to k výuce nebo k její vlastní vnitřní potřebě nikoli za účelem přímého nebo nepřímého obchodního nebo jiného hospodářského prospěchu.

Vlastní vnitřní potřebou *Masarykovy univerzity* se rozumí užití díla nejen v původní podobě, ale též ve zpracované nebo jinak změněné podobě zahrnující též takové užití mého díla touto vysokou školou, které spočívá v zadání mého školního díla k dalšímu zpracování jinému studentovi této vysoké školy (členovi téže akademické obce) za účelem vytvoření další kvalifikační práce nebo jiného školního díla, které bude odvozené od mého díla při uvedení mého autorství, názvu mého díla a pramene; a to vše v souladu s rozvojem vzdělanosti na *Masarykově* univerzitě a zájmem této veřejné vysoké školy navazovat na výsledky mé práce a mé školní dílo dále rozpracovávat v téže akademické obci.

Okolnosti hodné zvláštního zřetele z mé strany, například zájem o vlastní další rozpracování své kvalifikační práce na *Masarykově univerzitě* nebo jinde, jsem povinen sdělit této vysoké škole prostřednictvím studijního oddělení nejpozději při odevzdání kvalifikační práce.

V Brně dne 10. 12. 2018

podpis studenta

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.

Big thanks go 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, Janka, Hanka, Žofka, Vlastík, Mišo, Dominik, Stela, Ondra, Kristína, Grandma, Lukáš, 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 the 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 the Figure 2.1 to achieve higher degree of flexibility and modularity.

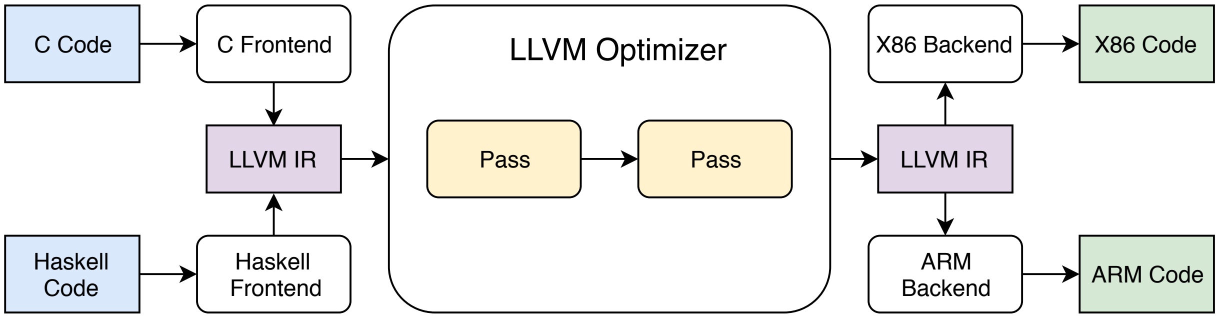

Figure 2.1: LLVM Compiler Architecture.

- 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]

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

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

```llvm
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
  %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.

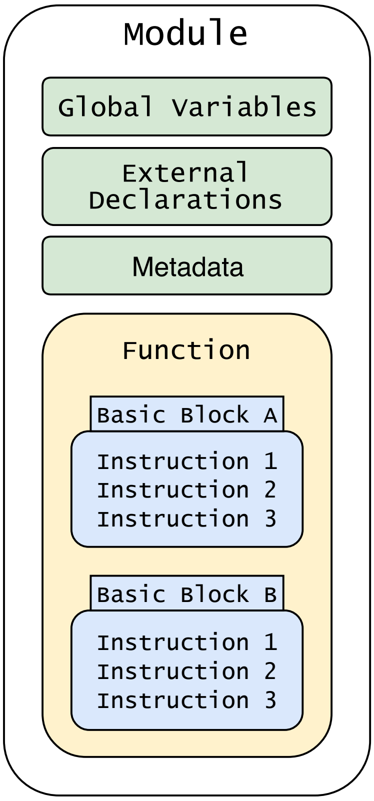

Figure 2.2: LLVM Module Structure.

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]

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

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 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. entry basic block is also prohibited to have any predecessors unlike for example if.then, which have entry as it processor 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, 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:
   - verify: Module Verifier.
   - view-cfg: View CFG of function.
   - instnamer: Assign names to anonymous instructions.

3. https://llvm.org/doxygen/classllvm_1_1TerminatorInst.html
4. https://llvm.org/docs/CommandGuide/opt.html

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

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

Getting the AST from the code above 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'
```

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

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

5. Syntactic structure of the source code represented in as a tree.
6. https://llvm.org/doxygen/namespacellvm.html

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
}
```

3 Extracting Program Subsets

In this chapter, we introduce the method for extracting parts of programs from the provided user input.

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

We follow with the example of the method from the user perspective in the section 3.2.

After example, we present in detail each major step that is followed. Starting with computing data dependencies graph (section 3.3). Following with the procedure for finding connected components in the computed data dependencies graph (section 3.4). Next follows introduction of the call graph (section 3.5) and subsequently procedure for finding path from source to target in it (section 3.6). The chapter ends with the section describing methods on eliminating dead components and functions from the code (section 3.7).

3.1 Method Overview

Mandatory **input** for the method is a touple with the following definition:

$$input \equiv (code, target)$$

where:

- **code** is a C program source code compiled into the llvm bitcode.
- **target** is an integer value representing line of code from the C program source code.

We also define **source** as an entry to the C program (`main` function).

The method determines what parts of the *input* to extract according to the *source* and *target*. Procedure subsequently calculates possible execution path up to the *target* and extracts this execution path into the separate, functioning executable.

Barring implementation specific details (which are discussed in the 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 call graph, mapping between connected components and functions that are being called from these components.
4. Find path from source to target in the call graph.
5. Eliminate dead components and functions that do not depend on the path.

Upon completion of the steps mentioned above, the llvm bitcode is produced as an *output*. We can define **output** as the extracted part of the original program according to the *source* and *target* while keeping the consistency of the code intact. By **consistency**, we mean that the *output* code is in a such state that it was possible to be compiled. *Output* is expected to be runnable the same way as the original. ****TODO? explain more here or in another chapter what it means for IR to be compiled without problems?****

3.2 Example

User provided us with the *input* in the form of the following C program source code that is stored in the file `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;
}
```

Since our method does not work directly with the C source code but instead works with the LLVM Intermediate Representation (IR), lets use clang and emit IR from the presented C source code in order to demonstrate the procedure more clearly: [1]

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

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.

Emitted LLVM IR is stored in the file `example.s` and has the following structure:[2]

example.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 @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 }
```

User also provided the line number **7** from the `example.c` as the target, which corresponds to the line **16** from the `example.s`. Source is the main function.

2. Strictly speaking, this is not exactly the IR code that would be emitted by the clang. We have stripped it out of the module info and comments to make it more readable. To see the unmodified `example.s`, please go to the Appendix B.

Procedure for finding mapping between C code referenced by the user input and its analogous IR instruction is implementation detail and is be described in the chapter 4.

When we apply the method on the contents of the `example.s` with respect to the source and target, we get the result stored in the file `example_extracted.s` with the following code:

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`, execution path from the program entry in the `main` function (which we will call **source**) to the **target** does not include function `foo` and its associated instructions, they can be removed. We are left only with function `bar` which contains target, and necessary instructions in the function `main` along with the `main` itself.

We can now take `example_extracted.s` and recompile it back into the functioning executable.[3]

3.3 Computing Data Dependencies

In order to identify what parts of the IR we can afford to remove, it is imperative to compute dependencies between instructions. When removing instructions, we need to preserve consistency of the remaining code so that it can be later compiled into a functional executable. To ensure this, we first compute dependencies among instructions.

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 due to the Marek Chalupa master's thesis *Slicing of LLVM Bitcode* [CHA25].

3. More about recompilation in the chapter 4.

- "**Control dependence** explicitly states what nodes are controlled by which predicate."
- "A **data dependence** edge is between nodes n and m iff n defines a variable that m uses and there is no intervening definition of that variable on some path between n and m. In other words, the definitions from n reach uses in m."

The crucial information comes from data dependencies. We need to make sure that the IR integrity will remain intact after we are done with removing IR instructions.

The following example demonstrates data dependencies 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
}
```

Taking `main` code, we can construct graph G where V is set of vertices (in our case vertex is instruction) and E is set of edges (in our case, edge between vertices V1 and V2 represents data dependency between instruction V1 and V2). In order to compute data dependencies, we use `dg` library. [CHA25] [4]

Computed **data dependency graph** for instructions from the function `main` is presented in the Figure 3.1. This graph is stored and used in the next step of the method for finding connected components (section 3.4).

Taking a closer look at the instruction: `%some_int = alloca i32, align 4`. We 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
```

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

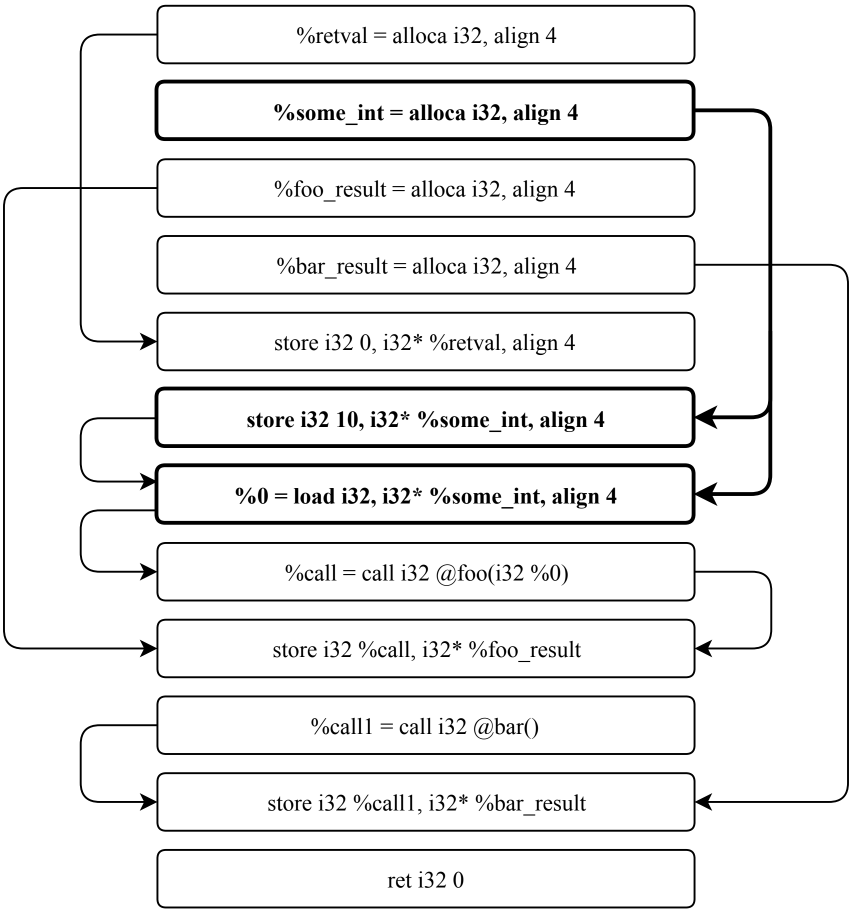

Figure 3.1: Data dependencies graph of the `main` function instructions.

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 and two dependent instructions would contain undefined values as their operand.[5] This would lead into unsuccessful recompilation of the modified code back into the executable.

3.4 Finding Connected Components

Having shown in the previous section what are inter-instruction data dependencies and how important they are in relation to the 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 whole picture. Since we have computed and stored graph of data dependencies, let us propose the idea of computing connected components of this graph.

5. 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

We define **connected component** as an isolated subgraph, where each pair of vertices is connected by some path.

We use **data dependency graph** computed in the section section 3.3 to find its connected components by using the following algorithm:

```
                         finding components
0. Let G = data dependency graph
1. Run Breadth-first search [Cor+09] on G to visit each instruction
2. IF instruction not in any component:
      Create new component and put instruction inside
   ELSE:
      Go to next instruction
```

Running the above mentioned algorithm on the **data dependency graph** produces connected components for each function in the input. As an example, we present components for the `main` function in the Figure 3.2 (for the clarity, each component has its own color).

Having instructions within each function separated into connected components comes 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

In respect to the program execution flow, user provided the target and we know the source. In order to proceed further, it is needed to compute possible paths from source to target that could be used by the execution of the program. Computing and walking the call graph of the program can produce for us this piece of information.

A **call graph**[6] is a control flow graph that represents relationship between program procedures in respect to control flow.[SSK09] Having call graph $G = \{V, E\}$, set of vertices V typically represents functions and set of edges E represents transfer of control flow from one function to another.

In our context, call graph represents relationship between individual connected components (computed in the section 3.4) and functions that are being called from these components by one of its instructions. In other words, our call graph is a set of mappings from components to the function (or functions).

We construct the call graph using the following algorithm:

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

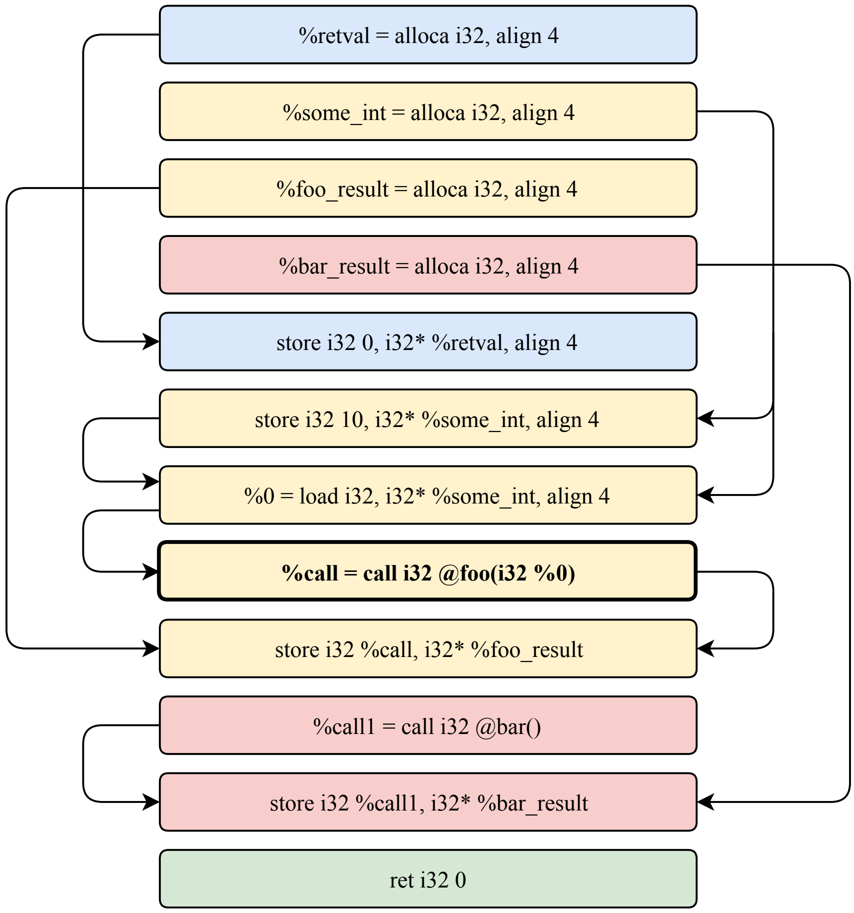

Figure 3.2: Connected components of the data dependencies graph for the `main` function instructions. Individual components are differentiated by the color.

```
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:
               Store information X gets called from the C
```

Running the presented algorithm on the **data dependence graph** of the `main` function that we computed earlier (Figure 3.2) produce call graph structure shown in the Figure 3.3. We know that in the context of the `main` function, there are four distinct components. We can see from the computed call graph, that `i32 @foo(i32 %n)` is being called from the *yellow* component, and `i32 @bar()` is being called from the *red* component.

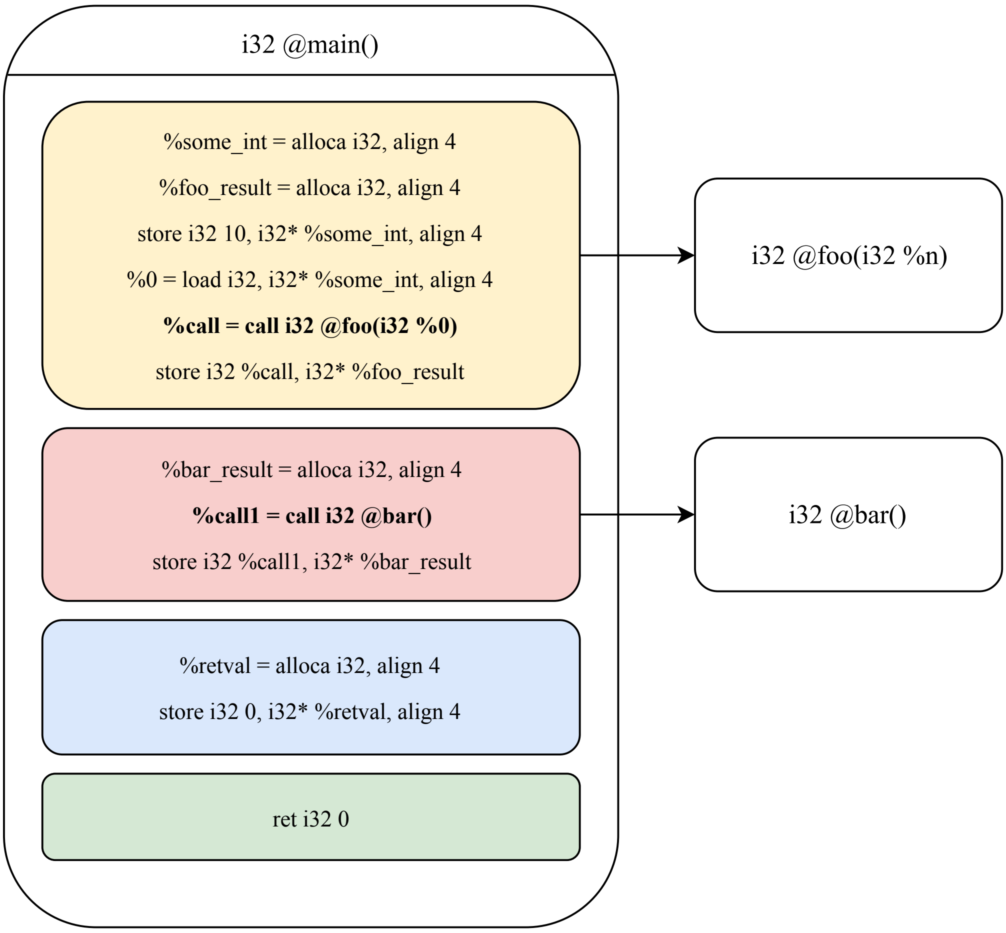

Figure 3.3: Call graph computed from the connected components as shown in the Figure 3.2.

3.6 Finding Path from Source to Target

Having call graph represented in the structure shown in the Figure 3.3 is beneficial for finding program execution flow path between specific components within the program in relation to their dependencies. We can find path from source to target and know which components this path contains.

*"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 non-adjacent otherwise."*[BM08] In our case, linear sequence of vertices consists of components computed in the section 3.4.

The reason why we constructed call graph in the previous chapter is now apparent. We want to find a path from source to the target and in doing so, know which connected components are part of this path or not. Potentially, there may exist infinite number of such paths. From the optimization standpoint, it would be fitting to find all (or at least as many as we can) paths and pick some path according to selected 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.

We will use the following algorithm in order to find a path from source to target in the call graph:

```
                              finding path
0. Let G = call graph
1. Run Breadth-first search on G to find path from source to target
```

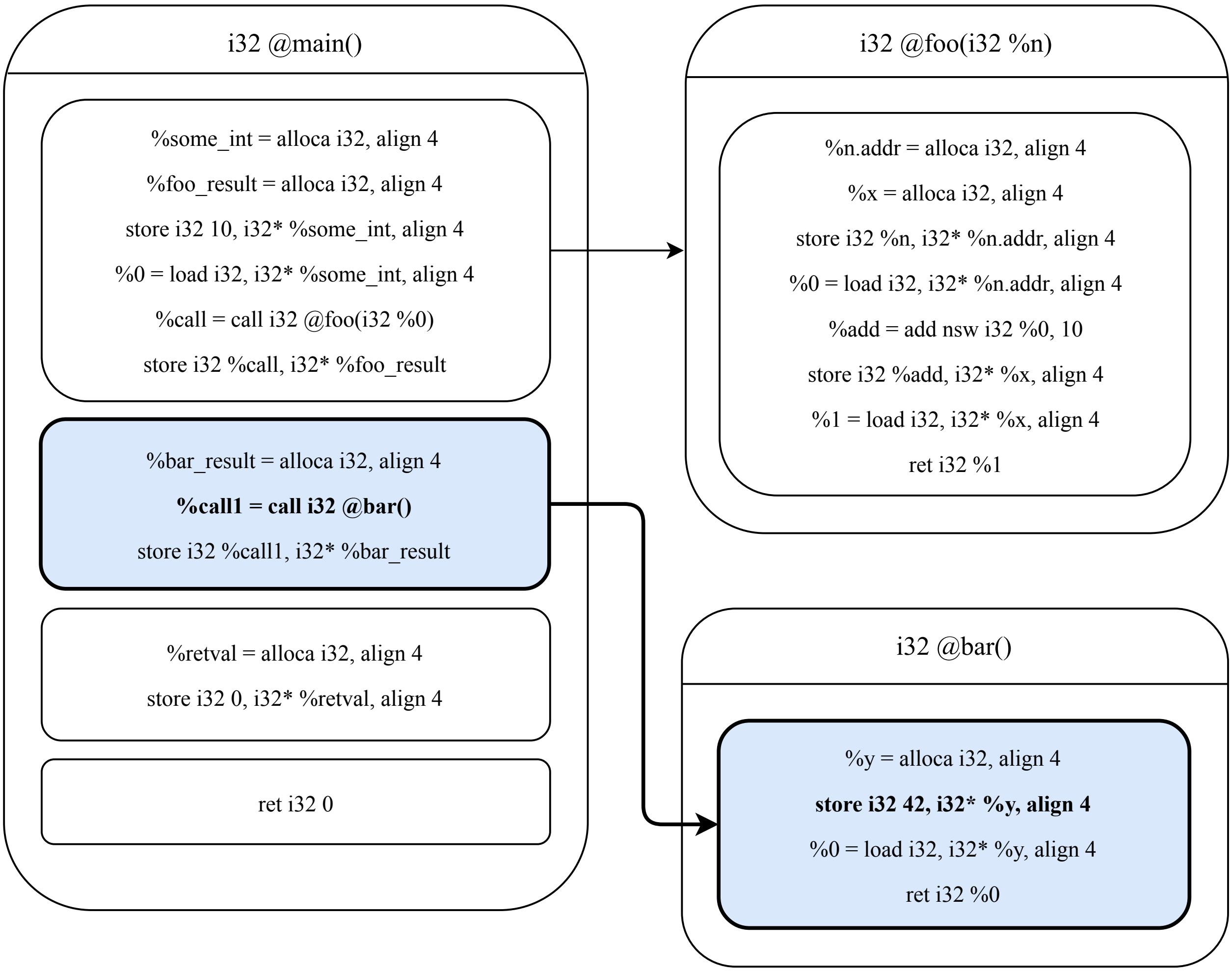

Figure 3.4: Path from **source** to **target** computed on the Figure 3.3. Components in the path are *blue*.

Given the source and target, we can see from the Figure 3.4 that path only contains two components, one from `main` function and another from `bar`. These two components are going to be the core of the final, extracted program.

3.7 Eliminating Dead Components and Functions

This section is the last step that the method performs. Components and functions that are dead are indentified and removed from the code and therefore, they will not be part of the final, extracted executable.

Element of the code marked as **dead** is defined as such, that it can be safely deleted without compromising integrity of the code. In other words, we can safely remove dead elements from the code without being worried that the execution starting from the source will not reach target.

Having successfully found path from source to target in the last section, we know that each component that is part of the path cannot be marked as dead. Therefore, components outside of path have to be explored and the decision found whether they can be marked as dead and removed or not.

We can 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 visualised in the Figure 3.5. We can see components marked as dead are colored *red* because they are not 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 terminator instruction `ret i32 0`.[7] Finally, all components marked as dead are removed and we are left with the code that we presented in the section 3.2 (`example_extracted.s`). Contents of the `example_extracted.s` are visualized in the Figure 3.6.

Unfortunately, the basic algorithm presented above works correctly only with the non-complex inputs that are similar to the `example.s`. We present two extensions of the `example.s` program in the subsection 3.7.1 and subsection 3.7.2 along with the improvements of the basic algorithm that cover these two extensions.

7. `https://llvm.org/doxygen/classllvm_1_1TerminatorInst.html` More about terminators and why they need special treatment in the chapter 4.

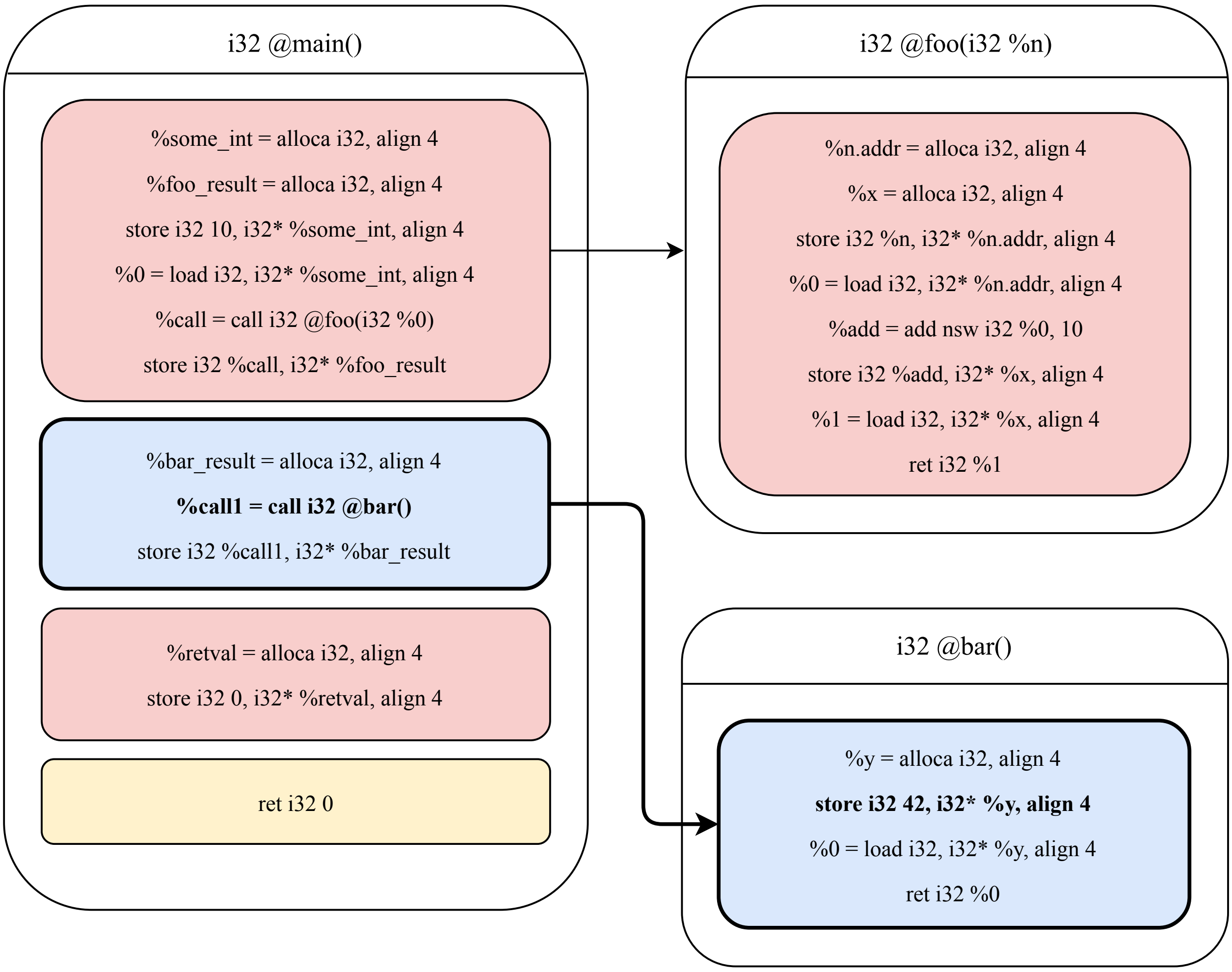

Figure 3.5: `example.s`: Components selected for removal are marked *red*. *Yellow* component is not marked for removal, because it contains terminator.

3.7.1 Path Depending on the External Function

We modify the `example.c` by introducing new function `int qux(void)`. Also, we change line 7 of the `example.c` from `int y = 42` to `int y = qux()` and save these modifications to the `example_mod1.c`:

example_mod1.c

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

int qux(void) {
    return 42;
}

```

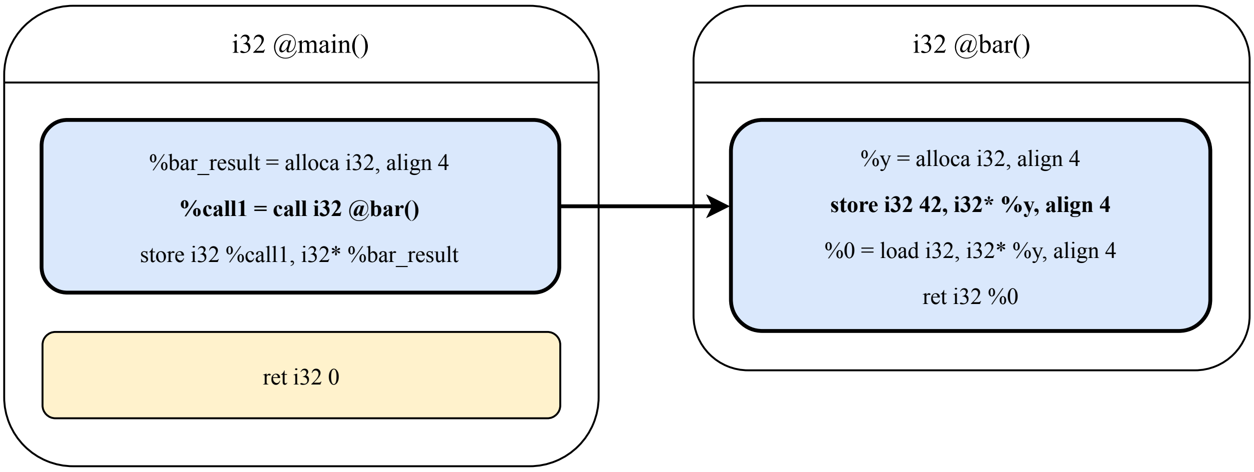

Figure 3.6: `example_extracted.s`: Final state after removing dead components and functions. `example.s`.

```
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 is the same (`int y = ...`) as in the original example from the section 3.2 (line 7 in the `example.c`). In the `example_mod1.c`, target sits at the line 11.

`example_mod1.c` compiled into the LLVM IR is stored in the `example_mod1.s`. Accordingly, the call graph with computed components and path for the `example_mod1.s` can be seen in the Figure 3.7.

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
```

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

example_mod1.s

```
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
}

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:
  %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
}
```

3.7.2 Path Depending on the Branching Instruction

To increase input complexity, lets take `example_mod1.c` and add branching inside main function as shown in the `example_mod2.c`. After taking look at the generated LLVM IR shown in the `example_mod2.s`, we can see branch instruction at the line 42. Also, there are present two new basic blocks that this branch instruction refers to: `if.then` and `if.end` (lines 44 and 49).

If we take a closer look at the generated call graph for `example_mod2.s` (Figure 3.9), 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`. This `if.then` basic block is handled by the branch instruction from the line 42. Therefore, algorithm will find component that contains this branch instruction and adds it to the path. The call graph with computed components and path that the algorithm takes as an input is shown in the Figure 3.9. Component in the main function marked as green contains branching instruction that the algorithm identified as needed and therefore added to the path.

The result of the algorithm are presented in the `example_mod2_extracted.s` and visually in the Figure 3.10.

```
example_mod2.c
1 int foo(int n) {
2     int x = n + 10;
3     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);
    int n = 10;
    if (n < 42) {
        int bar_result = bar();
    }
    return 0;
}
```

example_mod2.s

```
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
}

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:
  %retval = alloca i32, align 4
  %some_int = alloca i32, align 4
  %foo_result = alloca i32, align 4
  %n = 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
  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

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

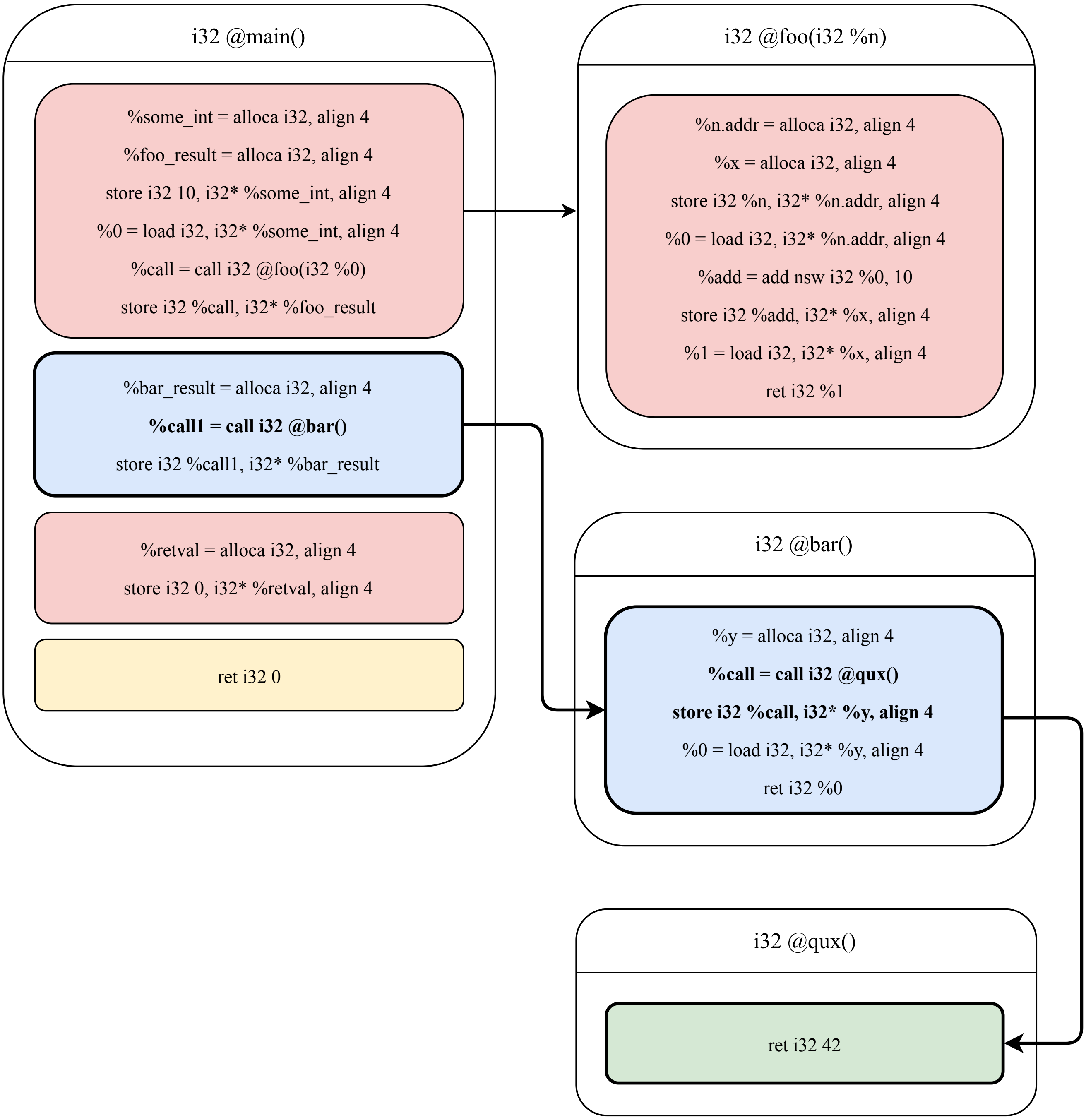

Figure 3.7: `example_mod1.s`: Components selected for removal are marked *red*. *Yellow* component is not marked for removal, because it contains terminator. *Green* component was discovered by the modified algorithm and has been added to the path and therefore will not be removed.

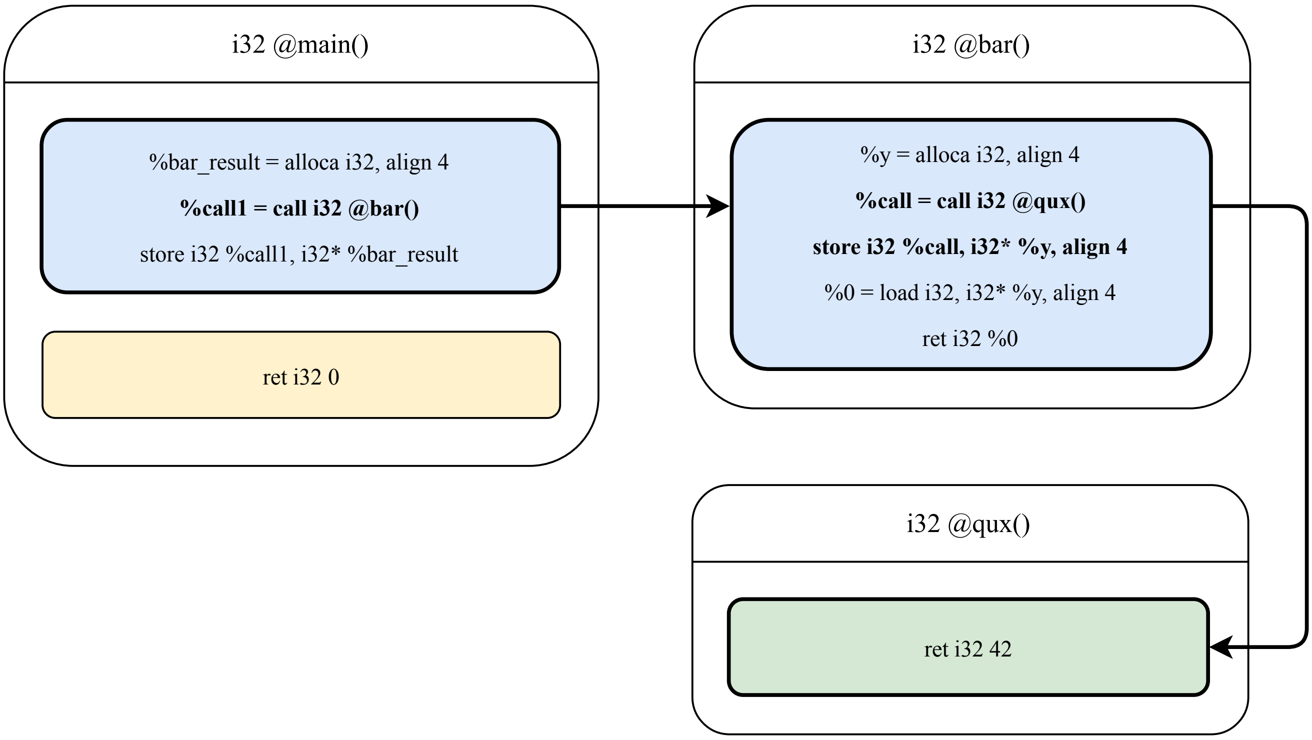

Figure 3.8: `example_mod1_extracted.s`: Final state after removing dead components and functions from `example_mod1.s`.

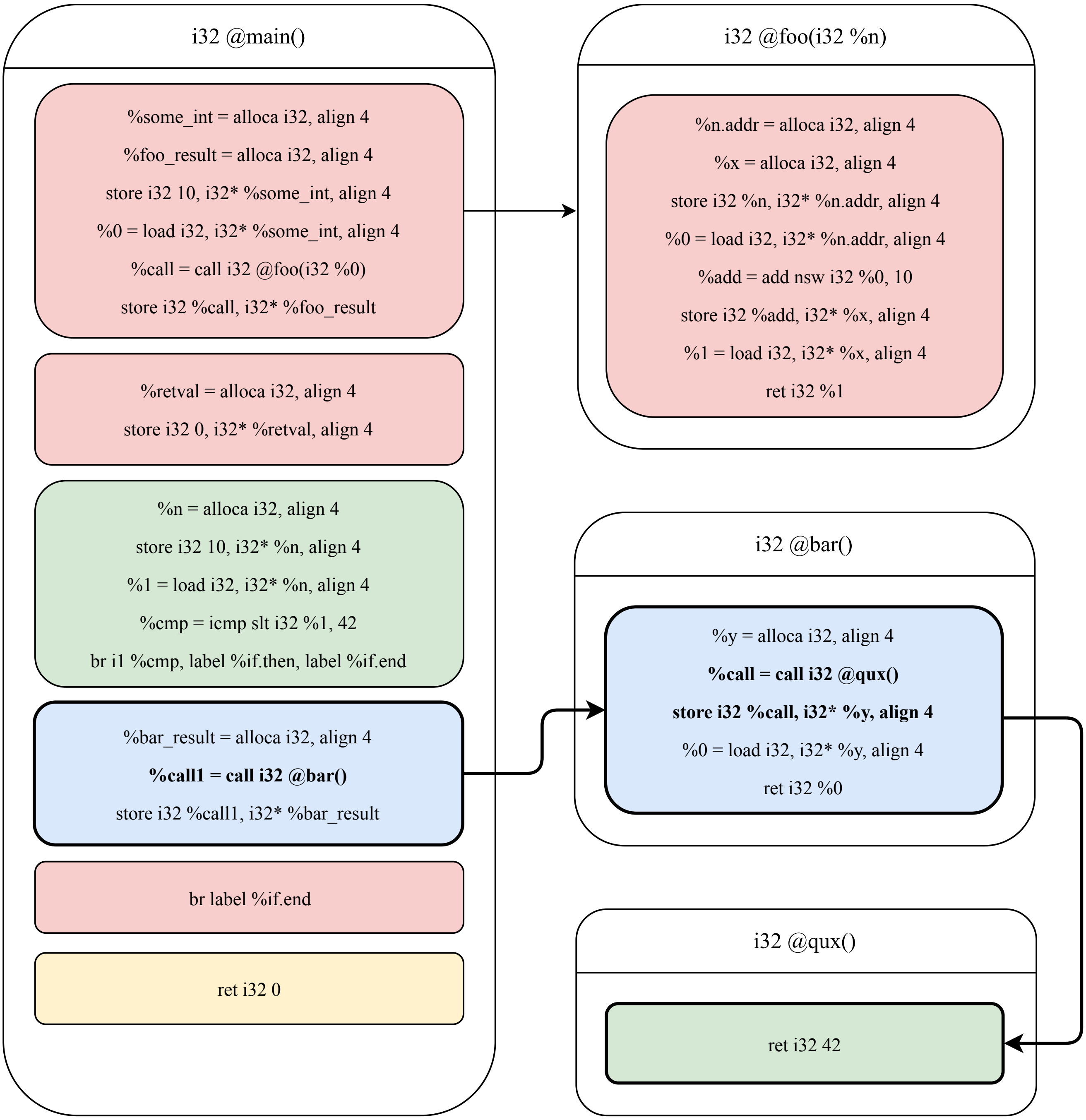

Figure 3.9: `example_mod2.s`: Components selected for removal are marked *red*. *Yellow* component is not marked for removal, because it contains terminator. *Green* components were discovered by the modified algorithm and were added to the path and therefore will not be removed.

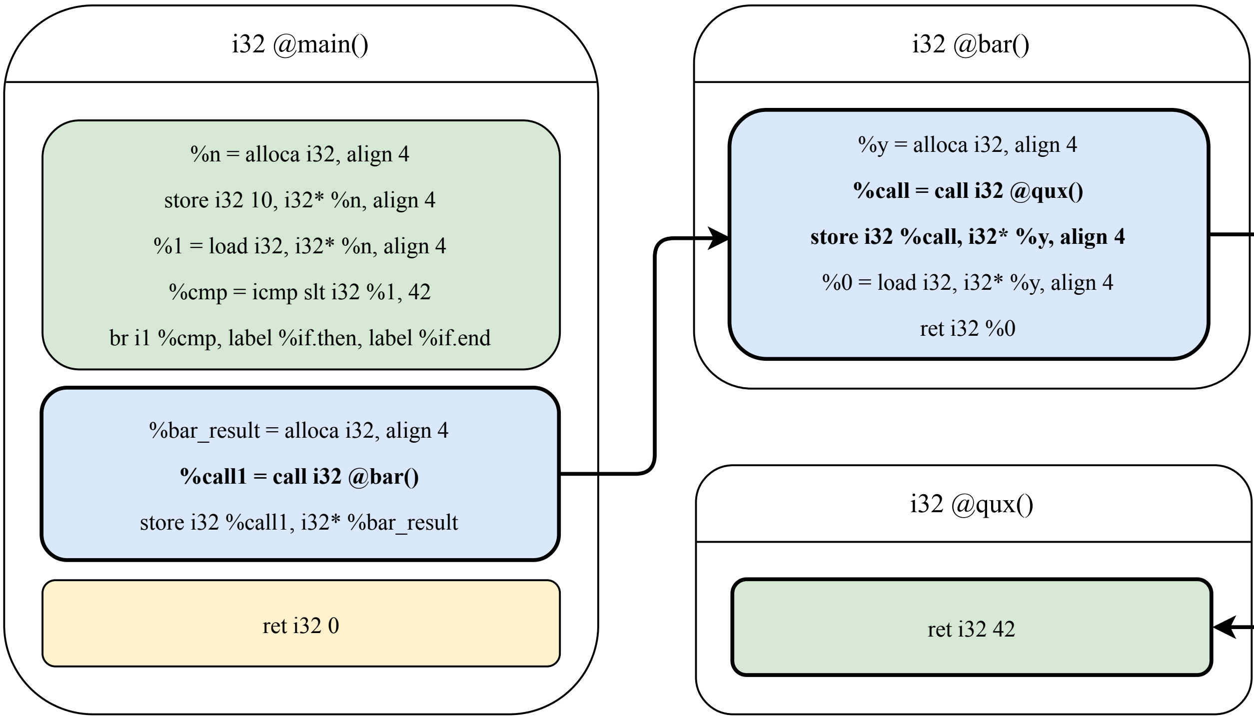

Figure 3.10: `example_mod2_extracted.s`: Final state after removing dead components and functions from `example_mod2.s`.

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 call graph, mapping between connected components and functions that are being called from these components.
  5. Find path from source to 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

TODO: examplain apex* functions and tell limitations

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 of 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:

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

example_extracted_final.s

```
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.

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:

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

```
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 python3. 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.

```
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 exected executable, apex.py 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: WIP

yes.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);
    int n = 10;
    if (n < 42) {
        int bar_result = bar();
    }
    return 0;
}
```

```
$ python apex.py examples/experiments/example_mod2/example_mod2.bc
 ↪  example_mod2.c 16
$ ./extracted
10
```

```
$ python apex.py examples/experiments/example_mod2/example_mod2.bc
 ↪  example_mod2.c 20
$ ./extracted
42

$ python apex.py examples/experiments/example_mod2/example_mod2.bc
 ↪  example_mod2.c 2
```

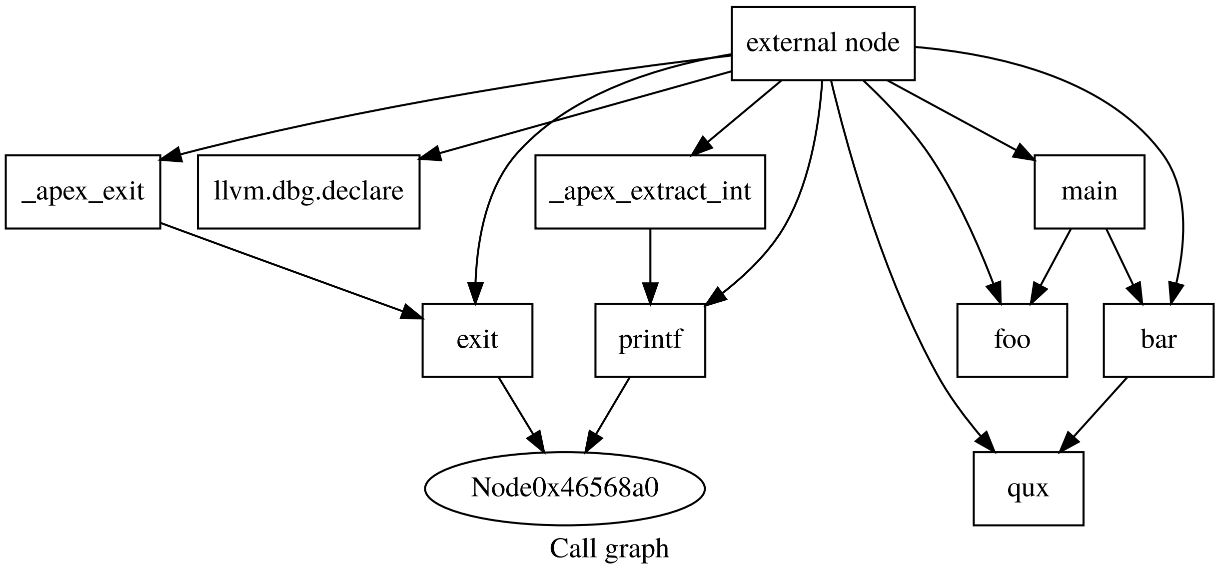

Figure 5.1: `example_mod2.c` After linking.

```
$ ./extracted
20
```

5.1 yes.c

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

We will use source code from the simple 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 following code:

yes.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define max(a,b) \
    ({ __typeof__ (a) _a = (a); \
     __typeof__ (b) _b = (b); \
     _a > _b ? _a : _b; })

int main(int argc, char* argv[]) {
    char* output;
    int output_size;
```

```c
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  }
```

First we need to compile yes.c into the LLVM bitcode with clang:

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

Lets pick as a target line 17 from the if branch: : `int needed_size = 2 + strlen(argv[1]);`

```
python apex.py examples/experiments/yes/yes.bc yes.c 17
./extracted test
> 6
./extracted foobar
> 8
./extracted
y
y
y
...
```

We can see that the extracted program outputs the value of `needed_size` when we provide command line argument. It also correctly executes the branch else when we do not prove any command line argument.

When we pick target line 31: `output_len = strlen(output);` and run apex.py, we get the following output when executing extracted binary:

```
python apex.py examples/experiments/yes/yes.bc yes.c 31
./extracted test
> 5
./extracted foobar
> 7
./extracted
y
y
y
...
```

Lets test the else branch of the code by picking target line 34: `output_len = 2;`

```
python apex.py examples/experiments/yes/yes.bc yes.c 31
./extracted test
test
test
test
...
./extracted
> 2
```

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

5.2 domainname.c

https://raw.githubusercontent.com/openbsd/src/master/bin/domainname/domainname.c

domainname.c

```
```

5.3 echo.c

6 Conclusions: TODO

6.1 Summary of the Results

6.2 Further Research and Development

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.

[Cor+09] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms, third edition*, 3rd. The MIT Press, 2009, ISBN: 0262033844, 9780262033848.

[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).

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

A Archive structure: TODO

Content of the attached archive:

TBA TBA TBA

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
```

```
  ret i32 0
}

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" }

!llvm.module.flags = !{!0}
!llvm.ident = !{!1}

!0 = !{i32 1, !"wchar_size", i32 4}
!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
```

```
  %foo_result = alloca i32, align 4
  %bar_result = alloca i32, align 4
  store i32 0, i32* %retval, align 4
  call void @llvm.dbg.declare(metadata i32* %some_int, metadata !28, metadata !12), !dbg !29
  store i32 10, i32* %some_int, align 4, !dbg !29
  call void @llvm.dbg.declare(metadata i32* %foo_result, metadata !30, metadata !12), !dbg !31
  %0 = load i32, i32* %some_int, align 4, !dbg !32
  %call = call i32 @foo(i32 %0), !dbg !33
  store i32 %call, i32* %foo_result, align 4, !dbg !31
  call void @llvm.dbg.declare(metadata i32* %bar_result, metadata !34, metadata !12), !dbg !35
  %call1 = call i32 @bar(), !dbg !36
  store i32 %call1, i32* %bar_result, align 4, !dbg !35
  ret i32 0, !dbg !37
}

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" }
attributes #1 = { nounwind readnone speculatable }

!llvm.dbg.cu = !{!0}
!llvm.module.flags = !{!3, !4, !5}
!llvm.ident = !{!6}

!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)
!1 = !DIFile(filename: "example.c", directory: "/mnt/Documents/work/university/muni/msc/thesis/APEX/examples/example")
!2 = !{}
!3 = !{i32 2, !"Dwarf Version", i32 4}
!4 = !{i32 2, !"Debug Info Version", i32 3}
!5 = !{i32 1, !"wchar_size", i32 4}
!6 = !{!"clang version 5.0.1 (tags/RELEASE_500/final)"}
```

```
!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)
!8 = !DISubroutineType(types: !9)
!9 = !{!10, !10}
!10 = !DIBasicType(name: "int", size: 32, encoding: DW_ATE_signed)
!11 = !DILocalVariable(name: "n", arg: 1, scope: !7, file: !1, line: 1, type: !10)
!12 = !DIExpression()
!13 = !DILocation(line: 1, column: 13, scope: !7)
!14 = !DILocalVariable(name: "x", scope: !7, file: !1, line: 2, type: !10)
!15 = !DILocation(line: 2, column: 9, scope: !7)
!16 = !DILocation(line: 2, column: 13, scope: !7)
!17 = !DILocation(line: 2, column: 15, scope: !7)
!18 = !DILocation(line: 3, column: 12, scope: !7)
!19 = !DILocation(line: 3, column: 5, scope: !7)
!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)
!21 = !DISubroutineType(types: !22)
!22 = !{!10}
!23 = !DILocalVariable(name: "y", scope: !20, file: !1, line: 7, type: !10)
!24 = !DILocation(line: 7, column: 9, scope: !20)
!25 = !DILocation(line: 8, column: 12, scope: !20)
!26 = !DILocation(line: 8, column: 5, scope: !20)
!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)
!28 = !DILocalVariable(name: "some_int", scope: !27, file: !1, line: 12, type: !10)
!29 = !DILocation(line: 12, column: 9, scope: !27)
!30 = !DILocalVariable(name: "foo_result", scope: !27, file: !1, line: 13, type: !10)
!31 = !DILocation(line: 13, column: 9, scope: !27)
!32 = !DILocation(line: 13, column: 26, scope: !27)
!33 = !DILocation(line: 13, column: 22, scope: !27)
!34 = !DILocalVariable(name: "bar_result", scope: !27, file: !1, line: 14, type: !10)
!35 = !DILocation(line: 14, column: 9, scope: !27)
!36 = !DILocation(line: 14, column: 22, scope: !27)
!37 = !DILocation(line: 16, column: 5, scope: !27)
```

D linked.ll

```llvm
; 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
}

```

```
declare i32 @printf(i8*, ...) #3

; Function Attrs: noinline nounwind optnone uwtable
define i32 @foo(i32 %n) #0 !dbg !25 {
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 !28, metadata !14), !dbg !29
  call void @llvm.dbg.declare(metadata i32* %x, metadata !30, metadata !14), !dbg !31
  %0 = load i32, i32* %n.addr, align 4, !dbg !32
  %add = add nsw i32 %0, 10, !dbg !33
  store i32 %add, i32* %x, align 4, !dbg !31
  %1 = load i32, i32* %x, align 4, !dbg !34
  ret i32 %1, !dbg !35
}

; Function Attrs: noinline nounwind optnone uwtable
define i32 @bar() #0 !dbg !36 {
entry:
  %y = alloca i32, align 4
  call void @llvm.dbg.declare(metadata i32* %y, metadata !39, metadata !14), !dbg !40
  store i32 42, i32* %y, align 4, !dbg !40
  %0 = load i32, i32* %y, align 4, !dbg !41
  ret i32 %0, !dbg !42
}

; Function Attrs: noinline nounwind optnone uwtable
define i32 @main() #0 !dbg !43 {
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
  call void @llvm.dbg.declare(metadata i32* %some_int, metadata !44, metadata !14), !dbg !45
  store i32 10, i32* %some_int, align 4, !dbg !45
  call void @llvm.dbg.declare(metadata i32* %foo_result, metadata !46, metadata !14), !dbg !47
  %0 = load i32, i32* %some_int, align 4, !dbg !48
  %call = call i32 @foo(i32 %0), !dbg !49
```

```
  store i32 %call, i32* %foo_result, align 4, !dbg !47
  call void @llvm.dbg.declare(metadata i32* %bar_result, metadata !50, metadata !14), !dbg !51
  %call1 = call i32 @bar(), !dbg !52
  store i32 %call1, i32* %bar_result, align 4, !dbg !51
  ret i32 0, !dbg !53
}

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" }
attributes #1 = { nounwind readnone speculatable }
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" }
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" }
attributes #4 = { noreturn nounwind }

!llvm.dbg.cu = !{!0, !3}
!llvm.ident = !{!5, !5}
!llvm.module.flags = !{!6, !7, !8}

!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)
```

```
!1 = !DIFile(filename: "src/apex/apexlib.c", directory: "/mnt/Documents/work/university/muni/msc/thesis/APEX")
!2 = !{}
!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)
!4 = !DIFile(filename: "example.c", directory: "/mnt/Documents/work/university/muni/msc/thesis/APEX/examples/example")
!5 = !{!"clang version 5.0.1 (tags/RELEASE_500/final)"}
!6 = !{i32 2, !"Dwarf Version", i32 4}
!7 = !{i32 2, !"Debug Info Version", i32 3}
!8 = !{i32 1, !"wchar_size", i32 4}
!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)
!10 = !DISubroutineType(types: !11)
!11 = !{null, !12}
!12 = !DIBasicType(name: "int", size: 32, encoding: DW_ATE_signed)
!13 = !DILocalVariable(name: "exit_code", arg: 1, scope: !9, file: !1, line: 6, type: !12)
!14 = !DIExpression()
!15 = !DILocation(line: 6, column: 21, scope: !9)
!16 = !DILocation(line: 7, column: 8, scope: !9)
!17 = !DILocation(line: 7, column: 3, scope: !9)
!18 = !DILocation(line: 8, column: 1, scope: !9)
!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)
!20 = !DILocalVariable(name: "i", arg: 1, scope: !19, file: !1, line: 11, type: !12)
!21 = !DILocation(line: 11, column: 28, scope: !19)
!22 = !DILocation(line: 12, column: 16, scope: !19)
!23 = !DILocation(line: 12, column: 3, scope: !19)
!24 = !DILocation(line: 13, column: 1, scope: !19)
!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)
!26 = !DISubroutineType(types: !27)
!27 = !{!12, !12}
!28 = !DILocalVariable(name: "n", arg: 1, scope: !25, file: !4, line: 1, type: !12)
!29 = !DILocation(line: 1, column: 13, scope: !25)
!30 = !DILocalVariable(name: "x", scope: !25, file: !4, line: 2, type: !12)
```

```
!31 = !DILocation(line: 2, column: 9, scope: !25)
!32 = !DILocation(line: 2, column: 13, scope: !25)
!33 = !DILocation(line: 2, column: 15, scope: !25)
!34 = !DILocation(line: 3, column: 12, scope: !25)
!35 = !DILocation(line: 3, column: 5, scope: !25)
!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)
!37 = !DISubroutineType(types: !38)
!38 = !{!12}
!39 = !DILocalVariable(name: "y", scope: !36, file: !4, line: 7, type: !12)
!40 = !DILocation(line: 7, column: 9, scope: !36)
!41 = !DILocation(line: 8, column: 12, scope: !36)
!42 = !DILocation(line: 8, column: 5, scope: !36)
!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)
!44 = !DILocalVariable(name: "some_int", scope: !43, file: !4, line: 12, type: !12)
!45 = !DILocation(line: 12, column: 9, scope: !43)
!46 = !DILocalVariable(name: "foo_result", scope: !43, file: !4, line: 13, type: !12)
!47 = !DILocation(line: 13, column: 9, scope: !43)
!48 = !DILocation(line: 13, column: 26, scope: !43)
!49 = !DILocation(line: 13, column: 22, scope: !43)
!50 = !DILocalVariable(name: "bar_result", scope: !43, file: !4, line: 14, type: !12)
!51 = !DILocation(line: 14, column: 9, scope: !43)
!52 = !DILocation(line: 14, column: 22, scope: !43)
!53 = !DILocation(line: 16, column: 5, scope: !43)
```